インターフェース増刊

MS-DOS プログラマーズ・バイブル

内部構造とシステム・コールとデバイス・ドライバと…

阿部英志 著

CXDB／Cソースレベル シミュレーションデバッガ

はじめに CXDBはWhitesmiths, Ltd.系のCクロスコンパイラを利用して作成した、プログラムのテストとデバッグを行なうためのCソースレベルシミュレーションデバッガです。
CXDBはより多くの環境で使用できるように、マンマシンインターフェースは扱いやすいウィンドウに設計されており、デバッグ対象プロセッサの実行をホストマシンの環境上でクロスシミュレーションします。
CXDBは対象プロセッサの完全なアドレス空間をアクセスすることができ、周辺装置やハードウェア割り込みのような無作為イベントのシミュレーションも可能です。
CXDBは起動時の指定で、スクロール機能付ウィンドウモードおよびラインモードのいずれかを指定できます。
そして、CXDBは(株)アドバンスド・データ・コントロールズが独自に開発した製品ですので、十分な技術サポートを伴って販売されます。

```
CXDB module:main.c
 57|   register char c_reg;
 58|   register short s_reg;
 59|   register long l_reg;
 60|   register int i_reg;
 61|
@62|   c_reg = retc( c_stat );
 63|   s_reg = rets( s_stat );
 64|   l_reg = retl( l_stat );
 65|   i_reg = reti( i_stat );
 66|   dnum = retd( 2.8 );
  #|   *c = retc( c_reg );

|0x0000011  e39000008ac
|0x00000128 4eb9000001ca       move.b  _c_stat,d7
|0x0000012e 1d47ffff           move.l  d7,(a7)
|0x00000132 3e39000008ae       jsr     _retc
|0x00000138 48c7               move.b  d7,(-1,a6)
|0x0000013a 2e87               move.w  _s_stat,d7
|0x0000013c 4eb9000001e0       ext.l   d7
|cxdb)                          move.l  d7,(a7)
                                jsr     _rets

| 1:text = 0x0
| 2:tsize = 0x576
| 3:data = 0x800
| 4:dsize = 0x110
| 5:stack = 0x1910
| 6:pc = 0x11e
| 7:sp = 0x10c8
| 8:time = 0x4b
| 9:x = 0x0
|10:x[0] = '#'
|11:adac[0] = 'A'
|12:c_stat = 0x11
|13:dnum =
|14:0.000000e+00
```

■コマンド入力
■コマンド実行結果の表示
一貫した単純なコマンドを用意しており、これらのコマンドをマクロとして自由に組み合わせ、定義して使用することにより複雑な処理を行うことが可能です。

ソースウィンドウ

■Cのソースコードの表示
■逆アッセンブルの表示
■反転➡現処理行
各種処理を行なうときの対象となる行を指す。
■@➡現実行行
プログラムカウンタが指すアドレスをもとにデバッグ情報から得た行を指す。
■*➡ブレークポイント行

ワッチウィンドウ

■指定した変数もしくはレジスタの内容を常時表示

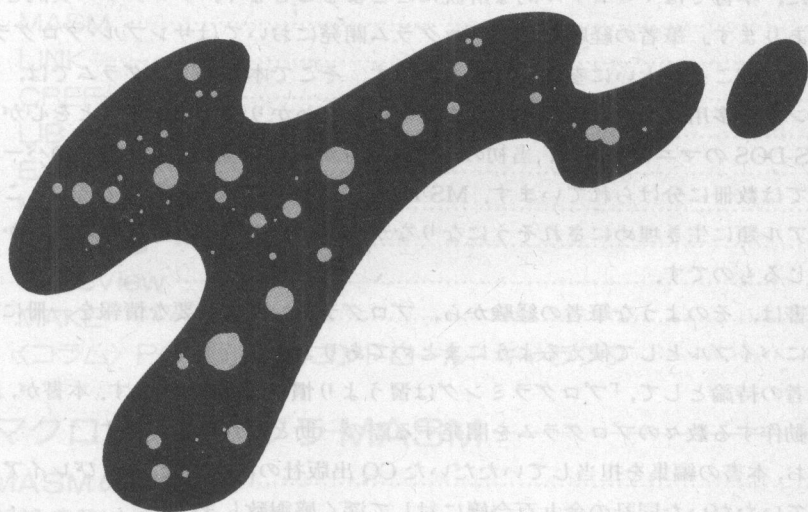
■コマンド一覧(その他に21コマンド有り)

A	命令の直接実行	I	機械語レベルのステップ・トレース
a	ラインアセンブラ	i	機械語レベルのステップ実行
B	ブレークポイントの表示	K	マクロの削除
b	ブレークポイントの設定	l	行・アドレス情報の表示
C	Cソース表示モード 注1)	M	マクロ登録表示
c	Cソースリストの表示 注2)	m	マクロ登録
D	全ブレークポイントの解除	me	マクロ登録、修正(ラインエディタ)
d	ブレークポイントの削除	N	次行ステップ実行
E	逆アセンブル表示モード 注1)	n	次行ステップ・トレース
e	逆アセンブルリストの表示 注2)	P	C・逆アセンブルマージモード 注1)
exit	コマンドの中断	p	C・逆アセンブルマージリスト表示 注2)
f	関数情報の表示	quit	CXDB終了
G	トレース	q	CXDB終了(確認付)
g	実行	r	エントリ関数の引数設定および再実行
h	コマンド履歴表示	S	ラインステップ実行

注1) ソースウィンドウへ出力　注2) コマンドウィンドウへ出力

MS-DOS プログラマーズ・バイブル

内部構造とシステム・コールとデバイス・ドライバと…

阿部英志 著

CQ出版社

はじめに

1982年にリリースが開始されたMS-DOSは，ここ数年の間に着々と機能の強化を行いながら発展してきました．MS-DOSは全世界で使用され，日本においても16ビット・パーソナル・コンピュータのOSとして90％以上のシェアをもつといわれています．

このような状況のもとで，MS-DOS上のアプリケーション・ソフトウェアの開発にしのぎを削るソフトハウスが数多く出現しました．そして，質・量ともに満足できるソフトウェアが開発されている現状に対して，幸せを感じているパソコン・ユーザは筆者だけではないでしょう．

しかし，MS-DOSの機能が強化され，アプリケーション・ソフトウェアの数や質が向上したといっても，MS-DOSを自分なりに(まさにパーソナル・コンピュータとして)使いこなすには，やはりプログラム開発のできる知識と環境が必要になってきます．

幸いにして，MS-DOSのプログラム開発環境(ツール)は，他のOSに比較して抜群に整備されているといっても過言ではないでしょう．すなわち使いやすく優れた市販ソフトが多いのです．したがって，残るはプログラム開発の知識(情報)の修得にかかわってきます．

本書は，MS-DOS上でプログラム開発を行う際に必要となる知識として，マクロ・アセンブラMASMを中心としたプログラム開発ツールの活用方法，およびMS-DOSの内部的な情報を網羅してあります．

また，本書ではマニュアル的な解説にとどまることなく，プログラム実例を豊富に掲げてあります．筆者の経験では，プログラム開発においてはサンプル・プログラム(ソース・リスト)こそが大いに参考になるものです．そこで本書のプログラムでは，日本語のコメントを多用してできるだけプログラム内容をわかりやすくすることを心がけました．

MS-DOSのマニュアルは，当初のバージョンから情報量が多く，現在のバージョンにおいては数冊に分けられています．MS-DOSのプログラムを開発する際に，これらのマニュアル類に生き埋めにされそうになりながら，プログラムを考えるのもなかなか苦痛を感じるものです．

本書は，そのような筆者の経験から，プログラミングに必要な情報を一冊に網羅し，まさにバイブルとして使えるようにまとめてあります．

筆者の持論として，「プログラミングは習うより慣れろ」があります．本書が，MS-DOS上で動作する数々のプログラムを開発する際の一助となれば幸いです．

なお，本書の編集を担当していただいたCQ出版社の清水当氏，およびレイアウトを担当していただいた同社の金丸百合嬢に対して深く感謝致します．

1989年　初秋

阿部英志

MS-DOS プログラマーズ・バイブル　目次

第I部　MS-DOSの構造とプログラム開発 ………… 9

第1章　MS-DOSの概要 ………… 10

1-1　MS-DOSの変遷 ………… 10
CP/Mコンパチ期 ………… 10
バージョン1.25から2.11へ ………… 10
バージョン3.10 ………… 11
各種のユーティリティ ………… 11
1-2　MS-DOSでのプログラム開発手順 ………… 15
エディット ………… 15
アセンブル/コンパイル ………… 15
ライブラリの作成/保守 ………… 16
リンク ………… 16
バイナリ・ファイル化 ………… 16
デバッグ ………… 17
1-3　プログラム開発ユーティリティ ………… 17
EDLIN ………… 18
MASM ………… 18
LINK ………… 19
CREF ………… 20
LIB ………… 20
EXE2BIN ………… 20
MAPSYM ………… 21
SYMDEB ………… 22
CodeView ………… 23
MAKE ………… 26
〈コラム〉PC-9801用のコントロール・キャラクタ ………… 17

第2章　マクロ・アセンブラMASM ………… 30

2-1　MASMの特徴 ………… 30
2-2　8086 CPUのセグメント ………… 30
セグメント方式のアドレッシング機構 ………… 30
セグメント方式の長所と短所 ………… 31
論理セグメント ………… 31
2-3　アセンブリ言語の記述 ………… 32
トークンとデリミタとセパレータ ………… 32
名前 ………… 34

ステートメント ……………………………………………………………………34
● nameフィールド(35) ● operationフィールド(35) ● operandフィールド(35) ● commentフィールド(35)

2-4 ディレクティブ(疑似命令) ……………………………………………………………35
モジュール・プログラミング ………………………………………………………40
● モジュール名(40) ● セグメントの定義(41) ● セグメント・レジスタの仮定(52) ● セグメントのグループ化(53) ● セグメントの簡略化定義(55) ● プロシージャの定義(58) ● PROCディレクティブによるパラメータ宣言(59)
ラベル ………………………………………………………………………………61
● LABELディレクティブによる定義(61) ● NEARラベルの定義(61)
外部参照 ……………………………………………………………………………61
データの定義と初期化 ……………………………………………………………64
● データの構造化(64)
マクロ機能 …………………………………………………………………………67
● マクロの定義(67) ● リピート・ディレクティブ(71)
条件アセンブル ……………………………………………………………………73
演算子 ………………………………………………………………………………76
● セグメント演算子(76) ● 型演算子(78) ● 数値演算子(79) ● 関係演算子(79) ● 演算子の評価順位(79)
〈コラム〉コメントのすすめ ………………………………………………………34
〈コラム〉前方参照 …………………………………………………………………58
〈コラム〉8086 vs 68000(その1) レジスタ ……………………………………73

第3章 MS-DOSの内部構造 ―――――――――――――――――― 82

3-1 MS-DOSのブート ………………………………………………………………82
ブート手順 …………………………………………………………………………82
● IPL(83) ● IO.SYS/MSDOS.SYSのロード(83) ● 初期化(84) ● CONFIG.SYS(84) ● COMMAND.COMのロード(84) ● AUTOEXEC.BATの実行(84)
config.sysファイル ………………………………………………………………84
● BREAKコマンド(85) ● DEVICEコマンド(85) ● FILESコマンド(85) ● BUFFERSコマンド(86) ● FCBSコマンド(86) ● LASTDRIVEコマンド(86) ● SHELLコマンド(86) ● COUNTRYコマンド(87)

3-2 プログラムのロードと実行 ……………………………………………………88
プロセスの起動 ……………………………………………………………………88
EXECシステム・コール …………………………………………………………89
プロセスの終了 ……………………………………………………………………94
PSP(Program Segment Prefix) …………………………………………………95
● PSPの構成(95)

環境変数と環境 …… 97
実行プログラムのメモリ・モデル …… 101
● EXE モデル(101) ● COM モデル(105)
〈コラム〉8086 vs 68000(その2) ストリング命令 …… 105

第4章 MS-DOS のファイル・アクセス —— 108
4-1 ファイル構造 …… 108
ディスク上の領域 …… 108
● 1Mバイト・フォーマットの場合(110) ● 640Kバイト・フォーマットの場合(110)
FAT とクラスタ …… 110
ディレクトリ・エントリ …… 111
● ディレクトリ・エントリの構造(112)
FAT …… 113
● 12/16 ビットの FAT 長(113) ● FAT の構造とファイルの関係(114)
階層ディレクトリの実現 …… 115
ディスク・バッファ …… 120
4-2 ファイル・ハンドルと FCB …… 121
FCB …… 122
● 基本 FCB の構造(122) ● 拡張 FCB の構造(123)
ファイル・ハンドル …… 124

第II部 MS-DOS のプログラム・インターフェース —— 125

第5章 MS-DOS の内部割り込み —— 126
5-1 内部割り込み …… 126
MS-DOS で使用できる内部割り込み …… 126
5-2 内部割り込みの機能と使いかた …… 126
〈コラム〉8086 vs 68000(その3) レジスタの保存 …… 132
〈コラム〉8086 vs 68000(その4) メモリ空間 …… 146

第6章 MS-DOS のファンクション・リクエスト —— 148
6-1 ファンクション・リクエストの種類と呼び出し方法 …… 156
INT21H による方法 …… 156
PSP を使用する方法 …… 156
CP/M 流の方法 …… 156
6-2 コンソール入出力関連のファンクション …… 156
6-3 外部入出力に関するファンクション …… 158

6-4　プロセス管理に関するファンクション……158
6-5　メモリ管理に関するファンクション……160
6-6　タイマ設定に関するファンクション……161
6-7　システム設定に関するファンクション……162
6-8　ディスク管理に関するファンクション……164
6-9　FCB 関係のファンクション……166
6-10　階層ディレクトリ管理に関するファンクション……169
6-11　ファイル・ハンドルに関するファンクション……170
6-12　デバイス制御に関するファンクション……175
6-13　MS-Networks に関するファンクション……180

第７章　システム・コール応用プログラム集――182

7-1　文字と文字列の操作……182
英小文字を大文字に変換する……182
● upper.c(182)　● uppersub.asm(183)
ASCII コードを表示する……183
● code.c(184)　● codesub.asm(185)
文字列を入出力する……185
● gstr.c(185)　● gstrsub.asm(186)
プリンタ/AUX 出力を行う……188
● praux.c(188)　● prauxsub.asm(188)
7-2　時刻・日付の操作……189
ストップ・ウオッチ……189
● keyin.c(189)　● keyinsub.asm(189)
日付と時刻を得る……193
● tm.asm(193)
DOS のバージョンを取り出す……196
● gver.c(196)　● gversub.asm(196)
7-3　メモリ/プロセスの操作……199
子プロセスを実行する……199
● child.asm(199)
メモリ・ブロックの操作を行う……204
● maloc.c(204)　● malocsub.asm(204)
除算エラーを検出する……210
● divset.asm(210)　● div0.asm(212)
割り込みエントリ・アドレスを表示する……214
● gint.c(214)　● gintsub.asm(215)
7-4　ディスク/ファイルの操作……217
ディスクのリセット/カレント・ドライブの変更を行う……217
● dres.c(217)　● dressub.asm(217)

ディスク情報を得る ……………………………………………………………219
● getd.c(219) ● getsub.asm(219)
FCB によるファイル・アクセスを行う ………………………………………224
● fcb.c(224) ● fcbsub.asm(228)
ファイルのタイム・スタンプを変更する ……………………………………233
● stmp.c(233) ● stmpsub.asm(236)
デバイス・データを調べる ……………………………………………………241
● gdev.c(241) ● gdevsub.asm(244)
メディア交換の可能性を調べる ………………………………………………248
● media.c(248) ● mediasub.asm(248)
7-5 ディレクトリの操作 ……………………………………………………………252
サブ・ディレクトリを作成する ………………………………………………252
● mkd.c(252) ● mkdsub.asm(253)
サブ・ディレクトリを削除する ………………………………………………254
● rmd.c(254) ● rmdsub.asm(256)
カレント・ディレクトリの表示/変更を行う ………………………………258
● chd.c(258) ● chdsub.asm(259)
階層ディレクトリを操作する …………………………………………………262
● d.c(262) ● dsub.asm(272)
〈コラム〉 BAT ファイルのネスト ……………………………………………192
〈コラム〉 MS-DOS 標準のコントロール・キャラクタ ……………………214
〈コラム〉 PC-9801 専用エスケープ・シーケンス ……………………………231
〈コラム〉 8086 vs 68000(その5) IBM が 68000 を採用 ……………248

第III部 デバイス・ドライバと拡張メモリ ―――― 279
第8章 MS-DOS のデバイス・ドライバ ―――― 280
8-1 構造と呼び出しの手順 ……………………………………………………………280
デバイス・ドライバの呼び出し ………………………………………………280
デバイス・ヘッダ …………………………………………………………………281
● リンク情報(281) ● デバイスの属性(281) ● ストラテジおよび割り込みルーチンへのポインタ(282) ● デバイス名またはユニット数(282)
コマンド・パケット ………………………………………………………………282
● レコード長(283) ● 論理装置コード(283) ● デバイス・コマンド・コード(283) ● ステータス(283)
8-2 I/O リクエスト・コマンド ………………………………………………………284
● INIT(284) ● MEDIA CHECK(ブロック・デバイス)(286) ● BUILD BPB(ブロック・デバイス)(286) ● READ, WRITE & VERIFY(286) ● NON-DETRUCTIVE READ & NO WAIT(キャラクタ・デバイス)(287) ● STATUS(キャラクタ・デバイス)(287) ● FLUSH(キャラクタ・デバイス)(288) ● DEVICE OPEN/CLOSE(288) ● REMOVABLE

MEDIA(ブロック・デバイス)(288) ● Generic IOCTL(288) ● Get/Set Logical Drive Map(288)
8-3 デバイス・ドライバのデバッグ方法 ……289
● デバイス・ドライバを組み込んでデバッグする(289) ● デバイス・ドライバをプログラムとしてデバッグする(289)

第9章 MS-DOSの拡張メモリ ―― 290
9-1 内部メモリの限界と拡張メモリ ……290
内部メモリの限界 ……290
拡張メモリ ……291
拡張メモリの動作 ……292
拡張メモリの利用方法 ……293
● EMMの存在を確認する(293) ● システムの環境を調べる(294) ● 拡張メモリの要求(294) ● 論理ページのマッピング(対応づけ)(294) ● マッピング情報の保存と復元(295) ● 拡張メモリの返却(296)
9-2 EMMファンクション ……296

Appendix A PC-9801シリーズのBIOS ―― 314
● キーボードBIOS(314) ● グラフLIO(315)

Appendix B グラフィック・コンソール・ドライバの作成 ―― 321
● グラフィック・コンソール・ドライバの処理(321) ● グラフィック・コンソール・ドライバの構成(321) ● seg.h(323) ● graph.asm(324) ● struc.h(327) ● graphc.cモジュール(328) ● glio.cモジュール(338) ● graphsub.asmモジュール(345) ● st.asmモジュール(348) ● gtestc.モジュール(351)
〈コラム〉8086 vs 68000(その6) セグメントの副作用 ……323
〈コラム〉ASCII制御コード ……327
〈コラム〉MS-DOS標準のエスケープ・シーケンス ……353

Appendix C RAMディスク・ドライバの作成 ―― 354
● RAMディスク・ドライバの構成(355) ● seg.h(355) ● ram.asm(356) ● struc.h(358) ● ram.cモジュール(360) ● ramsub.asmモジュール(369) ● st.asmモジュール(373)

参考・引用文献 ―― 377
索　引 ―― 378

〈表紙デザイン〉 アイドマ・スタジオ

第I部

MS-DOSの構造とプログラム開発

第I部では，MS-DOSアプリケーション・プログラムを開発する際に必要となる知識として，プログラム開発ユーティリティやマクロ・アセンブラの使いかた，およびMS-DOSの内部構造やファイル構造について解説します．

まず第1章では，MS-DOS上におけるプログラム開発の手順について解説しています．MS-DOSには，大部分のプログラム開発ユーティリティが標準で添付されており，プログラム開発を行うには，これらユーティリティの使用方法は完全にマスタしておく必要があります．

つぎに，第2章ではマクロ・アセンブラMASMを取り上げています．MS-DOSは，8086 CPUを対象としているため，8086 CPUの特徴であるセグメントに関する知識は必須のものとなります．このため，マクロ・アセンブラMASMでもセグメント管理に関するさまざまなディレクティブや演算子が用意されており，MS-DOS上のアプリケーション・プログラム開発には，これらセグメント関連のディレクティブや演算子は避けて通ることのできない知識の一つとなります．

第3章では，MS-DOSのプログラム構造について解説しています．MS-DOSでは，大きく分けて二つのメモリ・モデルを用意しており，ユーザがプログラム開発するに当たっては，これらメモリ・モデルのルールについてもよく理解しておかなければなりません．

第4章では，MS-DOSのファイル・システムについて解説してあります．一般にユーザ・プログラムでは，なんらかの形でファイル/デバイスとの入出力を行う必要があります．したがって，第3章のプログラム構造と同様に，このMS-DOSのファイル構造についても熟知しておく必要があります．

第1章 MS-DOSの概要

バージョンとコマンドとユーティリティ

この章では，MS-DOSプログラム開発の基礎的知識として，MS-DOSの各バージョンの相違や，MS-DOSプログラム開発ユーティリティの機能と，その使用方法について解説していくことにします．

1-1 MS-DOSの変遷

1971年にインテル社の開発によってマイクロコンピュータがこの世に登場して以来，早くも20年近くの歳月が流れてしまいました．当初のマイクロコンピュータは，制御用途における電子回路を置き換えるための便利な部品として重宝にされていました．しかし，TSS(Time Sharing System)用に開発されたBASIC言語が，MS-DOSの生みの親であるマイクロソフト社によってマイクロコンピュータに移植されると，マイクロコンピュータには“汎用コンピュータ”としての道が開かれることになります．

■ CP/Mコンパチ期

1970年代にBASICが主流となっていたマイクロコンピュータに，フロッピ・ディスクを取り付けて，ハードウェアに依存するソフトウェア部分をBIOS(Basic I/O System)にまとめてプログラムの移植性を確保し，各種の言語やアプリケーションを動作/発展させてきたCP/M(Control Program for Microcomputer)の開発は，マイクロコンピュータがのちにパーソナル・コンピュータとして発展を遂げる大きな原動力になっていることは周知のごとくです．

1980年代の初頭において，パーソナル・コンピュータの主流が16ビット機へと移行すると，CP/M-80をベースに開発されたCP/M-86とMS-DOSが登場します．MS-DOSも当初のバージョンであるver.1.25では，CP/M-86と大差なく，そのシェアも五分五分の感がありました．しかし，ver.2.11になってUNIXを指向した数々の先進的な機能を取り入れたことによって，16ビット機のソフトウェア・バスとしての主導権争いでは，MS-DOSが完全に主役の座についてしまいました．

MS-DOS ver.1.25は，アメリカでは1982年3月にリリースが開始され，日本では1年後に日本電気のPC-9801用としてリリースされました．このver.1.25では，まだ階層ディレクトリ構造やデバイス・ドライバの概念はなく，io.sysへのエントリはCP/Mと同様にジャンプ・テーブル方式となっていました．

またこのver.1.25では，io.sysが16Kバイト，msdos.sysが8Kバイト程度でOS自体のサイズも小さく，メイン・メモリが128Kバイト程度でも実用可能なサイズとなっていました．しかし，このver.1.25は，日本での流通期間が非常に短く，MS-DOSとして一般に普及したのはver.2.11になってからといえます．

■ バージョン1.25から2.11へ

ver.1.25に対して，ver.2.11では大幅な変更が加えられました．その主な変更点を列記すると，次のようになります．

(1) デバイス・ドライバの概念を導入し，io.sysをデバイス・ドライバの集合体としている．これにともない，config.sysファイルを用いてユーザによるデバイス・ドライバの追加が可能となった．

(2) 階層構造のディレクトリをサポートするようになった．

(3) 標準入出力に対して，他のデバイスやファイルをリダイレクトすることが可能となった．また，パイプ処理も可能となっている．

(4) システム・コールの数が増え，ファイル・ハンドルによるファイル・アクセスを可能とした．これによって，ファイルとデバイスを同格化して扱うことが可能になったばかりでなく，ファイル・アクセスに関する手続きが簡略化された．

(5) 子プロセス・コール機能が追加された．これによって，たとえばデバッガからエディタを呼んでソース・ファイルの修正を行ったり，エディタの中からコンパイラを呼んでコンパイルするなど，コマンドの階層的な使いかたが可能となった．

これらの大幅な変更にともない，ver.2.11ではio.sysが40Kバイト，msdos.sysが17Kバイトとなり，OSのサイズも大幅に大きくなりました．したがって，ver.2.11では，メイン・メモリが128Kバイトではマクロ・アセンブラさえも使用できなくなり，このころからメモリ・ボードの開発と価格競争に拍車がかけられることになります．

バージョン3.10

そして，ver.2.11に対してローカル・エリア・ネットワーク対応の機能追加を行ったver.3.10が登場するころになると，MS-DOSのマルチタスク版の噂も流れはじめ，1988年になって遂に(ようやく？)MS-DOSの兄貴分として，マルチタスクをサポートしたOS/2が市場に流れはじめました．

PC-9801用のMS-DOS ver.3.10は，1985年11月にリリースされ，バージョンが上がるにつれてしだいにシステムのサイズも膨れ上がり，ファイル上でio.sysが32Kバイト，msdos.sysが28Kバイトにもなりました．このサイズは，同じver.3.10でも，デバイス・ドライバの追加の際にコマンドによる追加/削除が可能になったバージョン(fainal版)では，io.sysが49Kバイト，msdos.sysが28Kバイトとなり，OS部分だけでも80Kバイト近くになっています．これにcommand.comや日本語FEP(Front End Processor)を加えると200Kバイトを楽に越えてしまうほどに巨大化しました．

このver.3.10では，ver.2.11に比較して次のような追加が行われています．

(1) デバイス・ドライバの属性ビットにOPEN/CLOSE/REMOVEビットが追加された．

(2) ファンクション・リクエストの種類が増えた．

(3) 12ビット単位だったFATが16ビット単位のFATも扱えるようになった．これによって，大容量ディスクの場合にクラスタ当たりのバイト数を小さくできるので，ディスクを効率よく使用することが可能となる．しかし，NECのver.3.10では，デバイス・ドライバが16ビットFATをサポートしていないので利用できない．

(4) ローカル・エリア・ネットワーク(MS-Networks)をサポートした．ただし，MS-Networksがver.3.10に含まれているのではなく，実際のネットワーク処理は，MS-Networksドライバ(別売)が行う．

したがって，ver.3.10はver.2.11に対してMS-Networksを動作可能とするため，MS-Networks用のシステム・コールの追加や，サーバとしてのディスク/プリンタおよびファイルの共有制御機能の追加，デバイス・ドライバの機能の追加など，MS-Networks用の環境修正を行ったものといえます．

このMS-Networksは，別売のハードウェアおよびソフトウェアを準備しなければならず，一種のアプリケーションよりの話になるため，MS-DOSの解説からは切り離して考えたほうがよさそうです．

また，今日のようにプリンタやハード・ディスクなどの周辺装置の価格が低下してくれば，特殊な場合を除いてネットワークを組む必要性も見あたりません．このような観点から，本書ではMS-Networksに関しての解説は割愛することにします．

さて，**表1-1**に示したようにMS-DOSは1981年の登場から着実に拡張を続けてきました．1983年のver.2.11でUNIXを指向した大改造を行い，1985年にはver.3.10でローカル・エリア・ネットワークへの対応を実現してきました．そして，ここにきてMS-DOSの640Kバイトの壁を破るため拡張メモリ機能を備えたver.3.30がリリースされました．

各種のユーティリティ

このようにMS-DOSは，16ビット・パソコンにおける標準OSとしての地位を不動のものとし，アプリケ

〔表1-1〕MS-DOSの変遷

リリース開始	バージョン	基本機能（拡張）	サブシステム	サイズ（標準ドライバおよびCOMMAND.COM含む）
1982年	1.25	CP/M-86互換		約24Kバイト
1983年	2.11	UNIX指向．階層ディレクトリ，入出力のリダイレクト，デバイス・ドライバなど		約74Kバイト
1985年	3.10	ローカル・エリア・ネットワーク対応	MS-Windows MS-Networks	約115Kバイト
1988年	3.30	拡張メモリ・ドライバのサポート	EMS 4.0	約133Kバイト

ーション・ユースから本格的なプログラム開発にいたるまで幅広くユーザのニーズに応えています．一説によると，MS-DOSは全世界で1000万台以上のパソコン上で動作し，この上で動作するアプリケーション・ソフトウェアは数万本にも達するといわれています．

しかし，MS-DOSが広く利用されているといっても，その機能がエンド・ユーザ・レベルで十分に活用されているとはいい難いようです．MS-DOSは，プログラミング環境としても優れたツールを提供しているハイレベルなOSとして位置づけられています．エンド・ユーザもこのMS-DOSのプログラム開発環境を利用しない手はありません．自分なりに使い勝手のよいコマンドを作成するなどして，大いに活用すべきです．

MS-DOS上でプログラム開発を行う場合，MS-DOSの機能を理解するとともに，プログラム開発用に提供されている各種のツールの機能を理解する必要があります．MS-DOSの機能とは，DOSの構造およびユーザに解放されているサービス・ルーチン(システム・コール)の利用方法，およびプログラムの構造などをいいま

〔表1-2〕MS-DOS ver.3.3の主要コマンド(PC-9801用システム・ディスク)①

コマンド名および構文	機　能	分　類	バージョン
ASSIGN [logicalunit=physicalunit]|/?	MS-DOSの論理装置に一つ以上の物理装置を割り当てる．また，ドライブ名を別のドライブに指定することも可能	外部(98)	2.1
ATTRIB [+R|−R] [+A|−A] file	ファイルの読み出し専用属性の設定/解除/表示を行う	外部	3.1
BACKUP {d1:|path|file} [d2:] [/S|/M|/A|/P|/D:date|/T:time|/L:file]	ハード・ディスクからフロッピ・ディスクに，一つまたはそれ以上のファイルのバックアップを行う	外部	3.1
BREAK [ON|OFF]	ctrl-Cチェック機能の設定/表示を行う	内部	2.1
CHDIR(CD) [d:] [path]	カレント・ディレクトリの変更/表示を行う	内部	2.1
CHKDSK {d:|file} [/F|/V]	指定されたドライブのディレクトリやFATを調べてディスクの状態を表示する．このときにメモリの状態も表示する	外部	2.1
CLS	画面クリア	内部	2.1
COMMAND [path] [device] [/P|/C string]	コマンド・プロセッサ	外部	2.1
COPY {path|file1} [/A|/B] {path|file2} [/A|/B|/V] または COPY file1+file2+…␣file	ファイルのコピーを行う．コピーはファイル単位でもディレクトリ単位でも可能．またファイルの連結を行うこともできる	内部	2.1
COPYA file AUX または COPYA AUX file	補助入出力装置(AUX)との間でファイル(データ)の送信/受信を行う	外部(98)	2.1
COPY2 [path] file [d:] または COPY2 file [path]/R	固定ディスク上のファイルをフロッピ・ディスクに退避する．または逆にフロッピ・ディスクから固定ディスクに復帰する	外部(98)	2.1
CTTY device	コマンドを入出力するデバイスを変更する	内部	2.1
CUSTOM [d:]/?	システム構築用のCONFIG.SYSファイルを会話形式で作成する	外部(98)	2.1
DATE [yy-mm-dd]	システムが管理する日付を設定/表示する	内部	2.1
DEL(ERASE) {path|file}	指定されたファイルの削除	内部	2.1
DIR [path|file] [/P|/W]	ディレクトリ内容の表示	内部	2.1
DISKCOPY [d1:] [d2:] [/V]	ディスク単位でのバックアップ・コピーまたはベリファイを行う	外部	2.1
DUMP {d:|path|file} [startaddress [endaddress]] [/D]|/?	ファイルの内容を16進数および文字で表示する．このとき，ファイル内における開始アドレス，終了アドレスも指定できる	外部(98)	2.1
EXIT	子プロセスとして起動されたCOMMAND.COMから親プロセスに戻る	内部	2.1
FIND [/V|/C|/N] "string" [file]	一つまたはそれ以上のファイルから指定した文字列を検索する	外部(フィルタ)	2.1
FORMAT [[d:] [/S|/V|/6|/9|/M|/P|/H]]|/?	指定されたドライブのディスクをMS-DOS用に初期化する	外部	2.1

す．

各種ツールの機能とは，アセンブラやコンパイラ，リンカなどのプログラム開発用ユーティリティの機能のことで，MS-DOS上のプログラムを自由に組めるようになるためには，これら開発ユーティリティの機能の理解は必須のものとなります．

＊　　　＊

なお，本書では特に断わりのない限りMS-DOS ver.3.30，およびマクロ・アセンブラ・パッケージver.5.1，MS-Cパッケージver.4.0を使用して，それらの内容に基づいてプログラミングや解説を行っています．

また，MS-DOSのバージョン番号は，OEMメーカによって多少の違いがあるため，本書ではNECから供給されているバージョン番号を用いています．

本書は，(個人レベルを含む)プログラマを読者の対象とし，コマンドの使用方法などは省略して表1-2にその一覧を掲げるにとどめておきます．

〔表1-2〕MS-DOS ver.3.3の主要コマンド(PC-9801用システム・ディスク)②

コマンド名および構文	機能	分類	バージョン
FC [/A|/B|/C|/L|/N|/T|/W] [/number] [/lbbuffsize] file1 file2	二つのファイル内容を比較し，その結果を表示する(ASCII比較またはバイナリ比較)	外部	2.1
HDUTL	ハード・ディスクの表面検査やスキップ・セクタの代替処理	外部	3.3
JOIN [d:path] [/D]	ディスク・ドライブを指定されたディレクトリに結合する．または，その結合の解除や結合状態の表示を行う	外部	3.1
KEY [[d:] path|file] [/S|/N]|/?	ファンクション・キーやカーソル移動キーに対して，ある機能の割り当てと取り消しを行う．また，その割り当て状況の表示を行う	外部(98)	2.1
LABEL [d:] [volumelabel]	ディスクのボリューム・ラベルの設定/変更/表示を行う	外部	3.1
MKDIR(MD) path	新しいディレクトリの作成	内部	2.1
MORE	標準入力から読み込んだ内容を1画面ずつ標準出力に出力する	外部(フィルタ)	2.1
MSASSIGN [olddrive=newdrive] [...]	ドライブ名を別のドライブ名に割り当てる．または，その割り当ての解除を行う	外部	3.1
PATH [;] [path1 [;path2] ...]	外部コマンドを検索するディレクトリ・パスの設定/解除/表示を行う	内部	2.1
PRINT [[/D:device |/B:buffsize |/U:busytick |/M:maxtick |/S:timeslice|/Q:quesize] [file] [/C|/P|/T|/R]	バックグラウンドでテキスト・ファイルなどをプリンタに出力する	外部	2.1
PROMPT [prompt text]	MS-DOSのコマンド・プロンプトの変更を行う	内部	2.1
RECOVER [d:|file]	不良セクタを含むディスクまたはファイルの修復を行う	外部	2.1
REN file1 file2	ファイル名の変更を行う	内部	2.1
RENDIR [d:] path1 path2	ディレクトリ名の変更を行う	外部	3.3
REPLACE [d:] {path1|file} [d:] {path 2} [/A|/D|/P|/R|/S|/W]	古いバージョンのファイルを新しいバージョンのファイルに更新する	外部	3.3
RESTORE d1:{d2:|path|file} [/S|/P|/B:date |/A:date |/E:time |/L:time|/M|/N]	BACKUPコマンドでバックアップしたファイルを復元する	外部	3.1
RMDIR(RD) path	ディレクトリの削除を行う	内部	2.1
SET [name=[parameter]]	環境領域に文字列を登録したり削除したりする．また，環境領域の内容の表示を行う	内部	2.1
SHARE [/F:filespace] [/L:locks] または SHARE /R	ファイルの共有やロックを行う	外部	3.1
SORT [/R] [/+n]	標準入力から読み込んだデータを並べ替えて標準出力に書き出す	外部(フィルタ)	2.1

〔表1-2〕MS-DOS ver.3.3の主要コマンド(PC-9801用システム・ディスク)③

コマンド名および構文	機能	分類	バージョン
SPEED [portnumber [parameters]]	システムに装備されているRS-232Cインターフェースのパラメータ設定とその起動を行う	外部(98)	2.1
SUBST [d:] [path] [/D]	パス名を仮想ドライブ名に置き換える．または，置き換えの解除/表示を行う	外部	3.1
SWITCH [RS232C-0 [parameters]] [PRINTER [parameters]] [MEMORY [parameters]] [COLOR [parameters]] [NDP1 [parameters]] [BOOT [parameters]] [NDP2 [parameters]]|/?	PC-9801のメモリ・スイッチの変更を行う	外部(98)	2.1
SYS d:	MS-DOSのシステム・ファイル(MSDOS.SYSとIO.SYS)を指定したディスクにコピーする	外部	2.1
TIME [hh [: mm [: ss]]]	システムが管理する時刻の設定/表示を行う	内部	2.1
TREE [d:] [/F]	指定されたドライブの階層ディレクトリ構造やファイル名の表示を行う	外部	3.3
TYPE file	ファイルの内容を表示する	内部	2.1
VER	MS-DOSのバージョン番号の表示	内部	2.1
VERIFY [ON|OFF]	ディスクへの書き込み時にベリファイを行うかどうかの設定/表示を行う	内部	2.1
VOL [d:]	ディスクのボリューム・ラベルの表示	内部	2.1
XCOPY [d1:path1] file1 [d2:path2] file2 [/A|/D:date|/E|/M|/P|/S|/V|/W]	ファイルやディレクトリ内容のコピーを行う．もし下位レベルのディレクトリが存在する場合には，それも含めてコピーを行う	外部	3.3

●バッチ・コマンド

コマンド名および構文	機能	分類	バージョン
ECHO [ON|OFF|message]	バッチ処理中にバッチ・ファイル内のコマンドを表示するかどうかを指定する．または引数で指定したメッセージの表示を行う	内部	2.1
FOR %% variable IN(set)DO command	バッチ処理におけるコマンドの反復実行を行う	内部	2.1
GOTO label	バッチ処理中に，指定したラベルに分岐する	内部	2.1
IF [NOT] condition command	バッチ処理中に条件判断によってコマンドの実行を行う	内部	2.1
PAUSE [remark]	バッチ処理の実行を一時的に停止する．このとき，引数で指定されたメッセージの表示を行う	内部	2.1
REM [remark]	バッチ処理中にメッセージの表示を行う．またはバッチ・ファイル内のコメント行を記述する	内部	2.1
SHIFT	バッチ処理中にコマンド行で指定された引数とバッチ・ファイル内のパラメータ(%0～%9)の対応関係を一つずつ左側にシフトする	内部	2.1

(1) 内部はcommand.comの内部コマンドであり，外部はファイルからロードされる外部コマンド．
(2) バージョン番号は，そのバージョン以降でサポートされているコマンドを表す(ver.1.xは区別していない．また機能拡張されたものは最新バージョン)．
(3) (98)はPC-9801固有のコマンド．
(4) (フィルタ)は，入出力をリダイレクト(またはパイプ指定)することによりフィルタとして使用可能なコマンド．
(5) 一般に，オプションは矛盾のない限り複数個の指定が可能．

1-2 MS-DOSでのプログラム開発手順

MS-DOSには，機械語を用いたプログラム開発のために，**表1-3**に掲げた各種のプログラム開発ユーティリティが揃っています．NEC版のMS-DOSの場合，これらのうち一部のユーティリティは，マクロ・アセンブラ・パッケージとして別売となっています．ここではプログラム作成の基礎知識として，同表のユーティリティを用いてプログラム開発を行う手順について簡単に説明しておきましょう．

一般に，MS-DOS上でアセンブラやコンパイラを使ってプログラム開発を行う際には**図1-1**のような開発手順を踏むことになります．

エディット

まずライン・エディタEDLINや市販のスクリーン・エディタ(最近ではDOS上で動くワープロを使う人も多いようである)などを使って，言語仕様にあったソース・プログラムを作成します．

このとき，アセンブリ言語であれば拡張子が“.asm”，FORTRAN言語であれば“.for”というように，言語によってデフォルトの拡張子が指定されている場合があるので，ファイル名の拡張子はこれらの言語にしたがって指定するようにします．

また，EDLINでは自動的にバックアップ・ファイル*.bakを作成してくれますが，ワープロなどを使う場合は，ユーザがあらかじめCOPYコマンドなどを使用して，このバックアップ・ファイルを作っておいたほうがよいでしょう．

アセンブル/コンパイル

次に，マクロ・アセンブラMASMやコンパイラなどを使ってソース・プログラムからオブジェクト・ファイル“.obj”を作成します．このオブジェクト・ファイルは，まだ実行可能なファイルにはなっていません．

ここで，類似した処理内容を含むプログラムの開発方法として二つの方法があります．一つは，どのプログラムでも類似した内容も含めてソース・ファイルを作成し，そのつど，そのファイルすべてをコンパイルする方法です．この場合は，後述のライブラリ・マネージャLIBによる処理は不要となります．

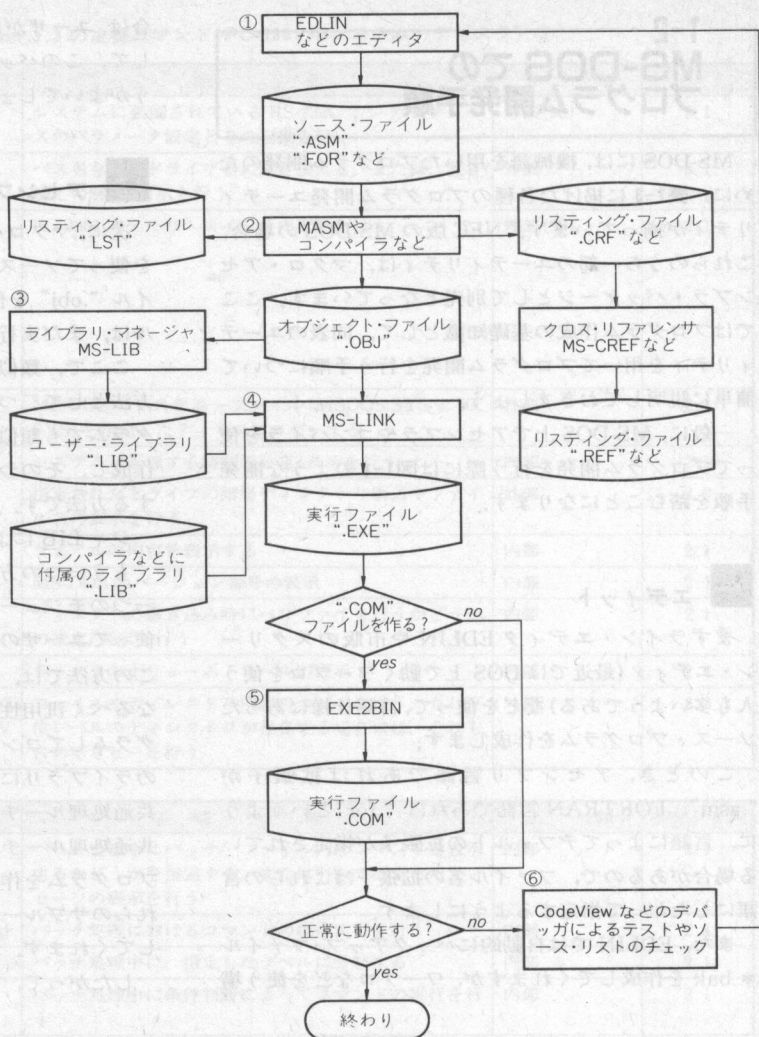
もう一つの方法は，よく使われる処理の部分だけを一つのモジュールとしてコンパイルし，これをLIBを使ってユーザのライブラリに登録しておく方法です．この方法では，たとえばある共通処理を行う部分を，なるべく汎用性のあるサブルーチンや関数の形にプログラムしてコンパイルしたのち，LIBを使ってユーザのライブラリに登録しておきます．そして，これらの共通処理ルーチンを必要とするプログラムから，その共通処理ルーチンを外部参照としてコールするようにプログラムを作成します．するとリンカLINKが，これらのサブルーチンや関数を検索して自動的にリンクしてくれます．

したがって，後者の方法を使えば，一つの完成され

〔表1-3〕
プログラム開発ユーティリティ

プログラム(ファイル)名	機能	バージョン	添付状況
EDLIN.EXE	ライン・エディタ	——	ver.3.30
MASM.EXE	マクロ・アセンブラ	5.10	マクロ・アセンブラ・パッケージ5.1
LINK.EXE	オーバレイ・リンカ	3.65	マクロ・アセンブラ・パッケージ5.1
CREF.EXE	クロス・リファレンサ	3.00	ver.3.10
LIB.EXE	ライブラリ・マネージャ	3.11	マクロ・アセンブラ・パッケージ5.1
EXE2BIN.EXE	ファイル変換ユーティリティ	——	ver.3.30
MAPSYM.EXE	シンボル・ファイル作成ユーティリティ	3.01	ver.3.30
SYMDEB.EXE	シンボリック・デバッガ	3.01	ver.3.30
CV.EXE	フルスクリーン・デバッガ	2.2	マクロ・アセンブラ・パッケージ5.1
MAKE.EXE	プログラム・メンテナ	4.07	マクロ・アセンブラ・パッケージ5.1

(1) バージョン番号は本書でテストしたバージョン番号．
(2) 添付状況で断りのないものはMS-DOSのバージョン番号．ただし，MS-DOSシステム・ディスクとマクロ・アセンブラ・パッケージの双方に添付されているものは，マクロ・アセンブラ・パッケージとして記した．

〔図1-1〕
MS-DOS上での
プログラムの開発手順

た共通処理のモジュールは，入出力のパラメータと機能をドキュメントにしておき，その中での処理についてはまったくのブラック・ボックスとして取り扱うことができます．これによって，プログラムの管理が楽になりプログラムの保守性が著しく向上することになります．

ライブラリの作成/保守

ライブラリ・マネージャLIBを使って，ユーザのライブラリを作成したり，既存のライブラリやコンパイラに付属のライブラリに新しくモジュールを追加したりする作業です．

また，これらのライブラリの中にあるサブルーチンや関数のモジュールを，ユーザの仕様にあったものと取り替えたりすることもできます．

リンク

次に，リンカLINKを使ってマクロ・アセンブラMASMやコンパイラなどによって作成されたオブジェクト・ファイル".obj"同士，あるいは，まえもって作成されたユーザ・ライブラリや，コンパイラに付属のライブラリの中にあるサブルーチンや関数のモジュールをリンクします．

リンクに成功すると，リンカLINKは，実行可能ファイル".exe"と，指定されていればメモリの割り当て情報の収められた".map"ファイルを出力します．

バイナリ・ファイル化

次に，64Kバイト以下のプログラム(COMモデル)であれば，EXE2BINコマンドを使ってバイナリ・ファ

イル“.bin”に変換することができます．

リンカで作成される“.exe”ファイルには，プログラム・ロード時のリロケート情報が含まれていて，プログラムを実行する際には，このリロケート情報によって MS-DOS がプログラムをメモリ上にリロケートするようになっています．

一方，EXE2BIN を通してバイナリ・ファイル“.bin”に変換されたプログラムは，COM モデルの基準(第 3 章参照)を満たしていれば，コマンド・ファイル“.com”にすることもできます．COM モデルでは，“.exe”ファイルのリロケート情報の部分が削除されるので，ファイルのサイズやロードの時間が短縮されることになります．

デバッグ

作成されたプログラムを実行してエラーがないかどうかをチェックする作業です．もし実行結果が正しくなければ，デバッガ SYMDEB や CodeView を使用してプログラムのデバッグ作業を行います．

この場合に，必要があればクロス・リファレンサ CREF を使用することにより，シンボルのクロス・リファレンス・リストを得ることができ，これによってデバッグがより効果的に行える場合もあるでしょう．

ソース・プログラムの記述に誤りが見つかったら，再びエディタを使用してプログラムの修正を行い，コンパイルやデバッグ作業を繰り返すことになります．

また，これらのアセンブル/コンパイルやリンク，あるいはデバッグなどの開発手順は，プログラム・メンテナ MAKE を用いることにより，必要な部分だけのアセンブルやコンパイルを自動的に行うことが可能となります．

*　　　*

以上の手順を経て，目的のプログラムが完成することになります．

1-3 プログラム開発ユーティリティ

MS-DOS のプログラム開発ユーティリティの起動方法と起動オプションについて，順番に解説していきます．一般に，MS-DOS の開発ユーティリティでは，起動方法として数種類の起動方法をもっています．

一つの方法は，ユーティリティ名を入力し，そのユーティリティで必要とするファイル名やオプションの

● PC-9801 用のコントロール・キャラクタ ●

MS-DOS では，ctrl キーと同時に各種のキーを押すといろいろな機能を実行することができ，このキー入力のことをコントロール・キャラクタと呼んでいます．

表 A は，MS-DOS に標準のものではなく，PC-9801 専用のコントロール・キャラクタの一覧です．

〔表 A〕PC-9801 専用のコントロール・キャラクタ

操　作	機　能
ctrl+F5	16 進によるデータ入力．ファンクション・キーの文字編集などに利用する
ctrl+F6	25 行表示と 20 行表示の切り換え(トグル)
ctrl+F7	ファンクション・キーの表示切り換え．1 回押すと F11 から F20 までの内容が表示される．2 回押すとファンクション・キーの表示は消え，すべての画面をユーザが利用できる
ctrl+F8	画面クリア(cls コマンドと同等)
ctrl+F9	画面の表示速度の変更．dir コマンドや type コマンドで画面表示する際にこのキーを押すと画面表示が遅くなる(トグル)
SHIFT+STOP	ctrl-S と等価で画面表示をストップする．ctrl-S でストップできないときでも有効

指定を会話形式で指定する方法です．また一つは，すべてのファイル名やオプションをコマンド・ラインに並べて起動する方法で，この方法はバッチ処理やMAKEファイルなどで利用しやすい方法です．ここでは，汎用性のある（自動実行が可能な）後者の方法について述べていきます．

EDLIN

EDLINは，テキスト・ファイルの編集を行うためのライン・エディタです．起動方法は次のとおりです．

EDLINの起動方法

```
EDLIN filename [/B]
```

filenameには，編集しようとするテキスト・ファイルのファイル名を指定します．filenameが，新しい名前の場合は新規にファイルが作成され，もし既存のファイル名の場合はそのファイルが読み込まれます．

EDLINは，テキスト・ファイルを操作の対象としているため，ファイルの途中でファイルの終端記号(End of File：EOF＝1AH)が検出された時点でファイルの読み込みを終了します．/Bオプションを指定すると，そのEOFを無視するため，ファイル内容の途中にEOFが入っているファイルの編集を行うことができます．**表1-4**にEDLINのサブ・コマンドの一覧を示しておきます．

なお，角カッコ［ ］はそのパラメータが省略可能なことを表し，本書ではすべての書式に対して角カッコのもつ意味を統一して表記しています．

MASM

マクロ・アセンブラMASMは，MS-DOS上の機械語プログラムを開発するためのアセンブラで，8086のファミリCPUやコプロセッサの命令ニーモニックがアセンブルでき，非常に強力で豊富な機能をもっています．

MASMの起動方法

```
MASM [options] sourcefile [, [objectfile]
[, [listingfile] [, [crossreferencefile]]]] [;]
```

sourcefileは，アセンブルするプログラムのソー

〔表1-4〕EDLINのサブコマンド

コマンド名	書　　式	機　　能
Append	[n] A	ディスク上のファイルからメモリ上のファイルへ〈n〉行読み込んで追加する
Copy	[行番号1]，[行番号2]，行番号3 [，回数] C	〈行番号1〉，〈行番号2〉で指定された範囲の行を，〈行番号3〉の直前へ指定された〈回数〉だけコピーする
Delete	[行番号1]，[行番号2] D	〈行番号1〉，〈行番号2〉で指定された範囲の行を削除する
Edit	[行番号]	〈行番号〉で指定された行を編集モードにする
Insert	[行番号] I	〈行番号〉で指定された行の直前にテキストを挿入する
List	[行番号1] [，行番号2] L	〈行番号1〉，〈行番号2〉で指定された範囲の行を表示する
More	[行番号1]，[行番号2]，行番号3 M	〈行番号1〉，〈行番号2〉で指定された範囲の行を〈行番号3〉の直前へ移動する
Page	[行番号1] [，行番号2] P	〈行番号1〉，〈行番号2〉で指定された範囲の行を23行までのページに分けて表示する
Quit	Q	編集中のファイルをディスクにセーブしないでEDLINを終了する
Replace	[行番号1] [，行番号2] [?] R 文字列1^Z [文字列2]	〈行番号1〉，〈行番号2〉で指定された範囲の行の中にある〈文字列1〉を〈文字列2〉と置き換える
Search	[行番号1] [，行番号2] [?] S [文字列]	〈行番号1〉，〈行番号2〉で指定された範囲の行の中にある〈文字列〉を検索する
Transfer	[行番号] T ファイル名	指定された〈ファイル〉を指定された〈行番号〉の直前に読み込んで挿入する
Write	[n] W	メモリ上のファイルから〈n〉行をディスク・ファイルに出力する
End	E	メモリ上のファイルをディスクにセーブしてEDLINを終了する

ス・ファイル名であり，デフォルトの拡張子は“.asm”となっています．objectfileは，アセンブルの結果が出力されるオブジェクト・ファイルのファイル名であり，デフォルトはソース・ファイル名のベース名に拡張子“.obj”を付けたものです．

listingfileは，アセンブル・リストを出力するファイル名でありデフォルトは“nul.lst”です．通常，ここにはソース・ファイル名と同じベース名を指定して拡張子“.lst”で出力させます．

crossreferencefileは，クロス・リファレンサCREFに入力させるクロス・リファレンス・ファイルのファイル名でありデフォルトは“nul.crf”です．ここでも，通常はソース・ファイル名と同じベース名を指定して，拡張子“.crf”で出力させたほうがよいでしょう．

optionsには，**表1-5**のようなオプションが用意されていて，これらのオプションを用いてMASMの機能選択を指定することができます．オプションは，下記のように環境変数“MASM”に指定しておくことにより，MASMを起動するたびに指定する手間を省略することも可能です．

```
set MASM=/A/ZI/Z
```

また，環境変数“INCLUDE”には，INCLUDEディレクティブによってファイルを取り込む際のディレクトリを指定することができます．

```
set INCLUDE=h:¥wk¥include
```

LINK

LINKは，個々に作成された8086のオブジェクト・コードのモジュール同士をリンクして，実行可能な“exe”リロケータブル・ファイルを作成するリンカです．

LINKの起動方法

```
LINK [options] objectfiles [,[executablefile]
[,[mapfile] [,[libraryfiles] [,[deffile]]]]] [;]
```

objectfilesには，LINKが入力すべきオブジェクト・ファイルのファイル名を指定します．オブジェクト・ファイルとは，MASMやコンパイラから出力されたファイルであり，一つまたは複数のファイル名を指

〔表1-5〕MASMの起動オプション

オプション	機能
/A	セグメントをアルファベット順に配置する
/B number	バッファ・サイズの設定．numberの単位はKバイトで1~63Kバイト．デフォルトは32Kバイト
/C	クロス・リファレンス・ファイルの指定
/D	アセンブル・リストにパス1におけるリスティングを追加する
/D symbol [=value]	シンボルの定義
/E	コプロセッサ・エミュレータ・ライブラリで使用するデータおよびコードの生成
/H	ヘルプ・メッセージの表示．コマンド・ラインの構文説明と，すべてのMASMオプションのリストを表示する
/I path	インクルード・ファイルのサーチ・パスを設定する
/L	アセンブル・リスティング・ファイルの作成を指定する
/LA	簡略化セグメント・ディレクティブの実行結果や，高級言語サポート機能によって生成されたコードをリスティング・ファイルへ出力する
/ML	すべての名前に関して，大文字/小文字を区別する
/MU	すべての名前を大文字に変換する
/MX	パブリック名と外部参照名に関して大文字/小文字を区別する
/N	リスティング・ファイルにおけるテーブルの作成を抑制する
/P	80286または80386特権モードで使用できない命令コードのチェックを行う
/S	セグメントをソース・コードに現われた順序で配置する
/T	アセンブルが正常に行われた場合のメッセージを抑制する
/V	アセンブルが正常に終了した場合のメッセージに加えて，処理した行数とシンボルの数も表示する
/W {0|1|2}	エラー表示(警告)レベルの設定
/X	条件アセンブルにおいて，条件が偽のためアセンブルされないブロックをリスティング・ファイルに出力する
/Z	エラーの発生したソース行をスクリーンに表示する
/ZD	行番号情報だけをオブジェクト・ファイルに出力する
/ZI	オブジェクト・ファイルにシンボリック情報と行番号情報を出力する

定することができます．

別々にコンパイルされた複数のモジュールをリンクしたい場合は，あらかじめライブラリに収めておいて，そのライブラリから検索するようにするのも一つの方法ですが，ライブラリに収めるほどのモジュールでない場合は，このプロンプトに対して，複数のモジュールのファイル名をスペースまたはプラス記号“＋”で区切って入力します．LINKは，特に指定のないかぎり，オブジェクト・モジュールのセグメントを見つけた順にリンクしていきます．

executablefileには，LINKから出力される実行形式プログラムのファイル名を指定します．デフォルトは，objectfilesで指定された最初のファイルのベース名になり，拡張子は“.exe”です．

mapfileには，MAPファイルのファイル名を指定します．LINKは入力(オブジェクト)モジュール内にある，各セグメントのアドレスや長さを示したMAPリストを拡張子“.map”のファイルに出力します．ユーザは，このMAPリストを見ることによって，各モジュールやセグメントがどのような配置でリンクされたかを知ることができます．このmapfileは，デフォルトで“nul.map”というファイル名になっています．MAPファイルが必要ない場合には，ファイル名を指定しないと何も出力されません．

libraryfilesには，コンパイラに付属のライブラリや，LIBで作成されたユーザ・ライブラリのファイル名を指定します．LINKは，外部参照されたサブルーチンや関数を指定されたライブラリから検索してピックアップし，これらをリンクして“.exe”ファイルとして出力します．

このlibraryfilesにも，複数のライブラリを指定することができ，その場合はファイル名をスペースか＋記号で区切って指定します．MS-Cなどある種のコンパイラでは，このlibraryfilesをobjectfilesの中に埋め込んで指定する場合もあり，その場合は，このlibraryfilesを改めて指定しなくても正しくリンク処理されます．

また，環境変数“LIB”にディレクトリやパス名を与えることによってもlibraryfilesを指定することができます．

```
set LIB=h:¥wk¥lib
```

optionsには，**表1-6**のようなオプションが用意されていて，これらのオプションを用いることによって，LINKの機能選択を指定することができます．このオプションも，環境変数“LINK”に対して指定することも可能です．

```
set LINK=/NOI/NO
```

CREF

CREFは，アセンブリ言語によって書かれたプログラム中のシンボル・リストを作成します．ユーザは，このリストを参照することによって，あるシンボルがどこで定義され，どこから参照されているのかを迅速に知ることが可能になっています．

CREFの起動方法

```
CREF clossreference [listingfile]
```

crossreferenceには，MASMから出力されたクロス・リファレンスのファイル名を指定します．ここで拡張子のデフォルトは“.crf”です．

listingfileには，出力されるリスティングのファイル名を指定します．このファイル名のデフォルトは，crossreferenceで指定したファイル名のベース名に拡張子“.ref”を付けたファイル名です．

LIB

LIBは，LINKで使用するライブラリ・ファイルの作成や修正を行います．

LIBの起動方法

```
LIB oldlibrary [/PAGESIZE:number] [commands]
[, [listfile] [, [newlibrary]]] [;]
```

oldlibraryには，操作しようとするライブラリのファイル名を指定します．拡張子のデフォルトは“.lib”です．指定したライブラリがない場合には，新しいライブラリを作成します．

/PAGESIZEオプションは，新しいライブラリのページ・サイズを決めたり，既存のライブラリのページ・サイズを変更するために使用されます．このページ・サイズのデフォルト値には16バイトが使用されます．

commandsには，モジュールの追加や削除などの操作を指定します．この際のコマンド・キャラクタは**表1-7**のようになっています．

listfileには，ライブラリ中にあるPUBLICシンボルのリストを収めるファイルのファイル名を指定します．

newlibraryには，内容の変更されたライブラリをディスクに書き出す場合のファイル名を指定します．このフィールドのデフォルトは，oldlibraryと同じファイル名が使用されます．

EXE2BIN

このユーティリティは，実行可能ファイル“.exe”をバイナリ形式のファイルに変換するコンバータです．

〔表1-6〕LINKの起動オプション

オ プ シ ョ ン	機 能
/HE [LP]	LINK のオプションの表示(この場合はファイル名の指定はしない)
/PAU [SE]	ディスクに書き込む際の一時停止(ディスク交換するときに用いる)
/I [NFORMATION]	リンク処理に関する情報(リンクの段階やオブジェクト・ファイル名など)の表示
/E [XEPACK]	EXE ファイルの中の繰り返された NULL キャラクタを取り除く
/M [AP] [: number]	すべてのパブリック・シンボルを mapfile に出力. number はパブリック・シンボルの上限値(デフォルト 2048)
/LI [NENUMBERS]	シンボルに関連するソース・プログラムの行番号情報を mapfile に出力する
/NOI [GNORECASE]	シンボルの大文字と小文字を区別する
/NOD [EFAULTLIBRARYSEARCH]	オブジェクト・ファイル内に指定されているライブラリの検索を行わない
/ST [ACK] : number	プログラムのスタック・サイズの指定. number はスタックのバイト数で 10 進整数
/CP [ARMAXALLOC] : number	プログラムがロードされるときに必要なメモリ・スペースの指定. number は 16 バイト・パラグラフ数で 10 進整数
/SE [GMENTS] : number	LINK で扱える論理セグメント数の指定. number は 10 進整数であり 1~3072(デフォルト 128)
/O [VERLAYINTERRUPT] : number	オーバレイを制御するための割り込み番号の指定(number は割り込み番号でありデフォルト 3FH)
/DO [SSEG]	マイクロソフト規約にしたがって, セグメント順序を割り当てる
/DS [ALLOCATE]	DS レジスタに, プログラム・データを含む一番低いデータ・セグメントを設定する
/HI [GH]	プログラムをメモリ内のできる限り高い位置に配置する
/NOG [ROUPASSOCIATION]	グループ指定の無視
/CO [DEVIEW]	CodeView で使用するデバッグ情報の埋め込み
/B [ATCH]	ライブラリやオブジェクト・ファイルが見つからなくてもパス名の入力を促さない
/F [ARCALLTRANSLATION]	far コールの最適化
/NOF [ARCALLTRANSLATION]	/F オプションを無効にする
/PAC [KCODE] : [number]	隣接するコード・セグメントをまとめる. number はグループの最大サイズを指定する(デフォルト 65530)
/NOP [AKCODE]	/PAC オプションを無効にする

〔表1-7〕
LIBのコマンド・キャラクタとその機能

キャラクタ	機 能
+	これに続く.obj ファイルをモジュールとしてライブラリに付加する
−	これに続く.obj モジュールをライブラリから削除する
*	これに続くモジュールを抜き出し, オブジェクト・ファイルにコピーする
;	残りのプロンプトに対してデフォルトの応答を指定する
&	現時点でのコマンド・ラインを拡張する. オペレーションが長い場合に使用する
CTRL + C	コマンドを中止する

前述のように“.exe”ファイルにはリロケート情報が含まれていますが, この EXE2BIN ユーティリティを用いてバイナリ・ファイルに変換することによって, リロケート情報の部分が削除され, ファイルのサイズやファイルのロード時間が短縮されることになります.

EXE2BIN の起動方法

```
EXE2BIN executablefile [binaryfile]
```

executablefile には, LINK から出力されたエラーのない“.exe”ファイルを指定します. ここで指定するプログラムは, ファイルの常駐部やファイル内のコード/データの部分は 64 K バイト以下でなければなりません. また, スタック・セグメントは存在してはいけません. このユーティリティで作成されたバイナリ・ファイルは, “.com”ファイルの基準を満たしていれば, binaryfile の拡張子を“.com”に指定することでコマンド・ファイルとして実行可能となります.

MAPSYM

このユーティリティは, デバッガ SYMDEB でシンボリックなデバッグを行うために必要となるシンボル・ファイルの作成を行います.

〔表1-8〕SYMDEB のサブ・コマンド ①

コ マ ン ド 名	構 文	機 能
Shell Escape	! 〈MS-DOS のコマンド〉	SYMDEB から MS-DOS のコマンドを子プロセスとして実行する
Comment	* 〈コメント〉	コメントを標準出力デバイスに出力．リダイレクト・コマンドとともに使用する
Source Line	.（ピリオド）	現在のソース・ラインの表示
Redirection	< 装置名 > 装置名 = 装置名	入力デバイスの変更 出力デバイスの変更 入出力デバイスの変更
Help	?	SYMDEB コマンドの一覧表示
Assemble	A [〈アドレス〉]	ニーモニックをアセンブルしてメモリに直接入れる
Break Point Set Break Point Clear Break Point Disable Break Point Enable Break Point List	BP [n] 〈アドレス〉 [〈回数〉] BC {〈リスト〉|*} BD {〈リスト〉|*} BE {〈リスト〉|*} BL	ブレーク・ポイントの作成 ブレーク・ポイントの削除 ブレーク・ポイントの一時的な無効化 ブレーク・ポイントの有効化 ブレーク・ポイントのリスト
Compare	C 〈レンジ〉 〈アドレス〉	メモリ内容の比較
Display	? 〈式〉	〈式〉の値の表示
Dump Ascii Dump Bytes Dump Words Dump Double words Dump Short reals Dump Long reals Dump Ten-byte reals Dump	DA [〈アドレス〉|〈レンジ〉] DB [〈アドレス〉|〈レンジ〉] DW [〈アドレス〉|〈レンジ〉] DD [〈アドレス〉|〈レンジ〉] DS [〈アドレス〉|〈レンジ〉] DL [〈アドレス〉|〈レンジ〉] DT [〈アドレス〉|〈レンジ〉] D [〈アドレス〉|〈レンジ〉]	メモリ内容の ASCII ダンプ メモリ内容のバイト単位の 16 進と ASCII ダンプ メモリ内容のワード単位の 16 進ダンプ メモリ内容のダブル・ワード（4 バイト）単位の 16 進ダンプ 4 バイト単位の浮動小数点数の表示（単精度） 8 バイト単位の浮動小数点数の表示（倍精度） 10 バイト単位の浮動小数点数の表示 直前のモードでのメモリ・ダンプ（デフォルト DB）
Enter Enter Bytes Enter Words Enter Double words Enter Short reals Enter Long reals Enter Ten-byte reals	E 〈アドレス〉 [〈値〉] EA 〈アドレス〉 [〈リスト〉] EB 〈アドレス〉 [〈値〉] EW 〈アドレス〉 [〈値〉] ED 〈アドレス〉 [〈値〉] ES 〈アドレス〉 [〈値〉] EL 〈アドレス〉 [〈値〉] ET 〈アドレス〉 [〈値〉]	メモリへバイト値を入れる 指定したアドレスに〈リスト〉中の文字の ASCII コードを入れる 指定したアドレスへバイト値を入れる 指定したアドレスへワード値を入れる 指定したアドレスへダブル・ワード値（4 バイト）を入れる 指定したアドレスへ短い浮動小数点数（4 バイト）を入れる 指定したアドレスへ長い浮動小数点数（8 バイト）を入れる 指定したアドレスへ 10 バイトの浮動小数点数を入れる

MAPSYM の起動方法

MAPSYM [/L] mapfile

mapfile には，LINK が出力する MAP ファイルのファイル名（デフォルト拡張子＝".map"）を指定します．MAP ファイルは，LINK に対して/MAP/LI オプションを指定することによって得ることができます．

MAPSYM ユーティリティによって，mapfile の拡張子を ".sym" にしたシンボル・ファイルが出力されます．

/L オプションは，プログラムに定義されているグループ名やプログラムの開始，および行番号の有無などに関する情報を表示させるものです．

SYMDEB

シンボリック・デバッガ SYMDEB は，MS-DOS ver.2.11 付属の DEBUG に対して，シンボリック・デバッグの機能や，ブレーク・ポイント機能などを強化したデバッガです．起動方法は次のとおりです．

SYMDEB の起動方法

SYMDEB[symbolfiles] [tergetfile[arguments]]

symbolfiles には，LINK から出力された MAP フ

〔表1-8〕SYMDEBのサブ・コマンド②

コ マ ン ド 名	構 文	機 能
Fill	F 〈レンジ〉 〈リスト〉	レンジのメモリをリストの値で埋める
Go	G [＝開始アドレス] [ブレーク・ポイント・アドレス]	デバッグ中のプログラムの実行
Hex	H 〈数値1〉 〈数値2〉	16進数の和と差の計算
Input	I 〈ポート〉	指定したポートから1バイト読み込む
Load	L [〈アドレス〉]|[〈ドライブ〉〈レコード〉〈カウント〉]	ファイルまたはディスクの論理レコードを読み込む
More	M 〈レンジ〉 〈アドレス〉	レンジで指定したメモリ・ブロックをアドレスへ移動(コピー)
Name	N [〈ファイル名〉] [〈引数〉]	ファイル名や引数のセット
Output	O 〈ポート〉 〈バイト〉	ポートへバイトを出力する
PTrace	P [＝開始アドレス] [〈カウント〉]	割り込みにも対応したトレース
Quit	Q	SYMDEBの終了
Register	R [〈レジスタ名〉] [〈数値〉]	レジスタ内容の表示
Search	S 〈レンジ〉 〈リスト〉	バイト値の検索
Set Source Mode	S {−|&|＋}	命令コードの表示形式の設定
Stack Trace	K [〈値〉]	スタック・フレーム内容の表示
Trace	T [＝〈開始アドレス〉] [〈カウント〉]	プログラムの実行とトレース
Unassemble	U [〈レンジ〉]	メモリ内容の逆アセンブル
View	V [〈アドレス〉]	指定したアドレスからソース・ラインを表示する
Write	W [〈アドレス〉 [〈ドライブ〉〈レコード〉〈カウント〉]]	ファイルまたはメモリ内容のディスクへの書き出し
eXamine Symbol map	X [*]	シンボル名とアドレス・リストの表示
	X? [〈マップ名〉!] [〈セグメント名〉:] [〈シンボル名〉]	
Open Map	XO [〈マップ名〉!] [〈セグメント名〉]	シンボル・マップまたはセグメントをセット
Symbol Set	Z 〈シンボル〉 〈値〉	シンボリック・アドレスに値をセットする

ァイルをMAPSYMユーティリティによって変換したシンボル・ファイル".sym"を指定します．ここで，symbolfilesには，複数のシンボル・ファイルを指定することが可能です．

tergetfileには，デバッグの対象となる実行ファイルの名前を指定します．

argumentsには，tergetfileに対して与える引数の指定を行うことができます．

表1-8に，SYMDEBのサブ・コマンドを一覧できるように整理しておきました．

CodeView

CodeViewは強力なフルスクリーン・デバッガで，アセンブリ・プログラムや高級言語プログラムの変数やスタックなどを，ソース・レベルで分析することが可能です．CodeViewを使ってソース・レベルのデバッグを行うには，アセンブル/コンパイルやリンク時にデバッグ用のオプションを指定しておかなければなりません．

CodeView用のコンパイル・オプションには，/Ziまたは/Zdオプションを指定します．また，場合によっては/Odオプションを必要とする場合もあります．CodeView用のリンク・オプションとしては，/COオプションを指定します．これによって，実行ファイル".exe"に対してシンボルやソース行に関する情報が埋め込まれます．起動方法は次のようになります．

CodeViewの起動方法

```
CV [options] executablefile [arguments]
```

optionsには，**表1-9**に示したオプションの中から一つ以上のオプションを指定することができます．

executablefileには，デバッグしようとする実行ファイル".exe"を指定します．

argumentsには，実行ファイルに対して与える引数を指定します．

CodeViewはスクリーン・デバッガであるため，マウスやキーボードを使って対話的にサブ・コマンドを実行していきます．**表1-10**に示したのは，CodeViewで用いるダイアログ・コマンド(キーボードから入力するコマンド)の一覧です．

〔表1-9〕CodeViewの起動オプション

オプション	機　　能
/B	モノクロ・モードの使用
/C　commands	CodeViewを起動したのちにcommandsで指定されたCodeViewのコマンドを自動的に実行する
/F	画面のフリップ(バッファを使用しない画面出力)の指定
/S	画面のスワップ(バッファを使用した画面出力)の指定
/M	マウス・ドライバが登録されていても，そのマウスを使用しない
/T	シーケンシャル・モードの指定
/W	ウィンドウ・モードの指定

〔表1-10〕CodeViewのダイアログ・コマンド ①

分　類	コマンド名	構　　文	機　　能
コードの実行	Trace	T　[count]	現在のソース行または命令を実行する．countは実行の回数
	Program Step	P　[count]	現在のソース行または命令を実行する．countは実行の回数(ルーチン，プロシージャ，割り込みはトレースしない)
	Go	G　[breakaddress]	現在のプログラム実行．breakaddressにはシンボル，行番号またはアドレスを指定
	Execute	E	現在のプログラムを低速で実行
	Restart	L　[arguments]	現在のプログラムを再開する．argumentsはプログラムに与える新しい引数
データと式の検査	Display expression	?　expression [，format]	シンボルまたは式の値を評価して表示する．expressionは任意の式であり，formatは表(h)の書式指定子(省略可)
	Examine Symbol	X X * X ?　[module!] [routine.] [Symbol] [*]	シンボルのアドレスを表示．module，routine，symbolでモジュール別，プロシージャ別，シンボル別に指定可．*はワイルド・カード
	Dump	D　[address|range] D　[type] [address|range]	メモリ内容の表示(サイズはデフォルト型) メモリ内容をtypeで指定された型で表示．typeは表(a)の型指定子
	Compare Memory	C　range　address	rangeによって指定されたメモリ・ロケーションにあるバイトと，addressから始まるメモリ・ロケーションの内容を比較する
	Search Memory	S　range　list	rangeで指定されたメモリ・ロケーションに対して，listで指定されたバイト値の検索を行う
	Port Input	I　port	portで指定されたハードウェア・ポートからバイトを読み込んで表示する
	Register	R	すべてのレジスタとフラグの現在値を表示する
	8087	7	8087シリーズ(コプロセッサ)のレジスタ内の現在の値を表示する
ブレーク・ポイントの操作	Breakpoint Set	BP　[address [passcount] ["commands"]]	addressはソース行，ルーチン名あるいはラベルで指定可．passcountには最初にブレーク・ポイントとなる回数を指定．commandsはダイアログ・コマンドの並びでありブレーク・ポイントが現われるたびに実行される
	Breakpoint Clear	BC　list BC　*	listで指定されたブレーク・ポイントをクリアする．*はワイルド・カードであり，すべてのブレーク・ポイントをクリアする
	Breakpoint Disable	BD　list BD　*	listで指定されたブレーク・ポイントを一時的に無効にする．*を指定すると，すべてのブレーク・ポイントが無効になる

〔表1-10〕CodeViewのダイアログ・コマンド ②

分　類	コマンド名	構　　文	機　　能
ブレーク・ポイントの操作	Breakpoint Enable	BE list BE *	listで指定されたブレーク・ポイントを有効にする．*を指定すると，すべてのブレーク・ポイントが有効になる
	Breakpoint List	BL	ブレーク・ポイントのルーチン，アドレス，行番号，有効/無効の状態などを表示する
Watch文の操作	Watch	W？expression［，format］ W［type］range	Watch文で指定した値は，プログラムの実行中にウォッチ・ウィンドウに表示される．expressionは，単純変数または変数と演算子の使用が可能．formatは，表(b)の書式指定子．typeは，表(a)の型指定子
	Watchpoint	WP？ expression［，format］	Watch文で指定されている式の値を監視し，expressionが真(ゼロ以外)になるとプログラムの実行をブレークする．formatは表(b)の書式指定子
	Tracepoint	TP？ expression［，format］ TP ［type］［range］	指定した式（expression）またはメモリ範囲(range)の値に変更があるとプログラムの実行をブレークする．formatは表(b)の書式指定子．typeは表(a)の型指定子
	Watch Delete	Y number Y *	ウォッチ・ウィンドウ内のWatch文(numberで指定)を削除する．*はワイルド・カードであり，すべてのWatch文の削除
	Watch List	W	シーケンシャル・モードでWatch文の評価を行って表示
コードの検査	Set Mode	S ［+|−|&］	コードの表示モードの設定 ＋：ソース・モード －：アセンブリ・モード ＆：混合モード
	Unassemble	U ［address|range］	addressで指定した位置からrangeで指定した範囲を逆アセンブルする
	View	V ［expression］ V ［.［filename：］linenumber］	テキスト・ファイルの指定された行を表示する(expressionはアドレス)filenameを指定すると指定したファイルをロード(linenumberでソース行を指定)
	Stack Trace	K	プログラムの実行中に呼び出したルーチンの表示
コードまたはデータの変更	Assemble	A ［address］	8086ファミリの命令(ニーモニック)をアセンブルし，addressで指定されたアドレスに格納する
	Enter	E ［type］address［list］	addressで指定されたメモリ領域に，listで指定されたデータ列をtypeで指定された型で格納する．typeは表(a)の型指定子
	Fill Memory	F range list	rangeで指定された範囲のメモリをlistで埋める
	Move Memory	M range address	rangeで指定されたメモリ・ブロックの内容をaddressで指定したメモリ・ブロックにコピーする
	Port Output	O port byte	portで指定されたハードウェア・ポートにbyteで指定されたバイト・データを出力する
	Register	R ［registername［［=］expression］］	registernameで指定したレジスタの内容を表示し，そのレジスタ内容をexpressionで指定した値に変更する．registernameとexpressionを省略すると，すべてのレジスタ内容の表示だけを行う
システム制御コマンド	Help	H	ヘルプ・メッセージの表示
	Quit	Q	デバッガCodeViewを終了してMS-DOSに戻る

〔表1-10〕CodeView のダイアログ・コマンド ③

分　類	コマンド名	構　　文		機　　能
システム制御コマンド	Radix	N	[radixnumber]	入力基数を radixnumber で指定された値に変更する．radixnumber を省略すると，現在の基数を表示
	Redraw	@		CodeView 画面の再表示(ウィンドウ・モードのみ)
	Screen Exchange	¥		出力画面の切り換え
	Search	/	[regularexpression]	正規表現(一つ以上の異なる文字列にマッチするパターン)をソース・ファイル内から検索する
	Shell Escape	!	[command]	CodeView から，command で指定された MS-DOS のコマンドを起動する
	Tab Set	#	number	TAB コードに対して number で指定された個数の空白コード(20 H)を埋め込む(デフォルトは 8)
	Option	O	[option [+|−]]	表(c)のオプションを表示したり，オプションの ON(+)または OFF(−)を指定する．option には，表(c)のオプションを指定．option を省略するとすべてのオプションの状態を表示する
入出力のリダイレクション	Redirection	<	devicename	コマンド入力を devicename から行う．devicename にはファイルも指定可
		[T]	> [>] devicename	CodeView の出力を devicename で指定したデバイスまたはファイルに行う．T はオプショナルであり，出力を CodeView 画面にエコーする．">>"で指定すると出力ファイルにアペンド・モード(追加)で書き込む
		=	devicename	CodeView の入力と出力を devicename で指定したデバイスに切り換える
	Comment	*	comment	comment をコメントとしてファイルに出力する
	Delay	:		リダイレクトしたファイルからの実行を 0.5 秒単位で遅らせる．1 行に複数の Delay コマンドを設定可能
	Pause	"		リダイレクトしたファイルからの実行を一時中止して，キーボード入力を待つ

(1) [　] の中は省略可

(2) address にはソース行やルーチン名，あるいはラベルなどのほか，次の構文によって完全なアドレスの指定が可能．

[segments:] offset

segments は省略可能であり，8086 ファミリ CPU のセグメント・レジスタを指定できる．

(3) range は次の二つの構文のうちから選択できる．

① startaddress endaddress

② startaddress L count

count はオブジェクトの数でありバイト数ではない(型指定に依存)．

MAKE

MAKE ユーティリティは，プログラム保守ユーティリティであり，プログラム開発の自動化を行います．MAKE ユーティリティの一般的な起動方法は次の構文によります．

MAKE の起動方法

MAKE [options] [macrodefinitions] filename

options には，**表1-11** のようなオプションが用意されていて，これらのオプションを用いることによって，MAKE の機能選択を指定することができます．

macrodefinitions には，MAKE ファイル内で定義されたマクロ値に対して，コマンド・ライン上で新しい値を指定します．

filename には，MAKE ファイルのファイル名を指定します．

MAKE は，ソース・ファイルやオブジェクト・ファイルに変更があった場合に，関連するファイルの再コンパイル/リンク処理などを自動的に実行し，それらのファイルの更新を行います．

〔表1-10〕CodeViewのダイアログ・コマンド ④

型	指定子	サイズ
Default	なし	デフォルト型(直前に指定した型)
Bytes	B	バイト
ASCII	A	ASCII文字
Integers	I	2バイト整数(16ビット符号つき10進数)
Unsigned Integers	U	2バイト整数(16ビット符号なし10進数)
Words	W	ワード(2バイト:16ビット16進数)
Double Words	D	ダブル・ワード(4バイト:32ビット16進数)
Short Reals	S	短い浮動小数点数値(4バイト:32ビット実数)
Long Reals	L	長い浮動小数点数値(8バイト:64ビット実数)
Ten-Byte Reals	T	10バイト浮動小数点数値(80ビット実数)

(a) 型指定子(type)

指定子	出力書式
d	符号つき10進整数
i	符号つき10進整数
u	符号なし10進整数
o	符号なし8進整数
xまたはX	16進整数
f	符号つき浮動小数点
eまたはE	符号つき科学技術表記
gまたはG	符号つき浮動小数点(f)または科学技術表記(eまたはE)のうち，より簡潔な値
c	単一文字
s	文字列(ASCIZ:文字列の最後が00H)

(b) 書式指定子(format)

option	対応するオプション
F	Flip/Swap
B	Byte-Coded
C	Case-Sense
3	80386

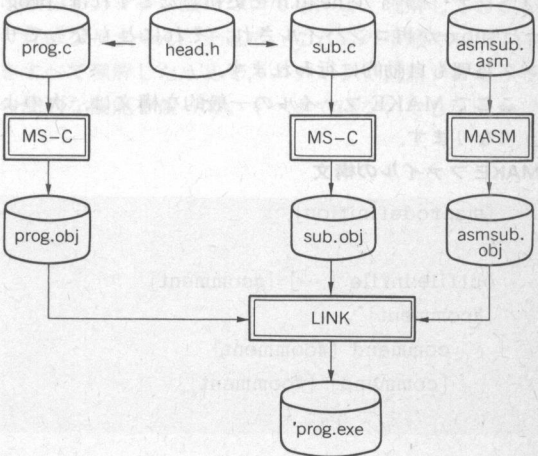
(c) option

MAKEでは，実行に際してファイルの関連づけや生成の手順を指定したファイルをあらかじめ作成しておかなければなりません．このファイルの関連や作成手順などの情報を収めたファイルを"MAKEファイル"と呼んでいます．

一般に，小さなプログラムの開発では，1本のソース・ファイルにすべてのプログラムを記述することになります．しかし，たとえばC言語で記述したプログラムの特殊なサブルーチン(関数)をアセンブリ言語で記述したような場合は，複数のソース・ファイルが必要になります．また，大規模なプログラムの開発では，プログラムのモジュール化(構造化)が重要であり，それにともなって一つのプログラムを多数のソース・ファイル(モジュール)に分けて記述することになります．

プログラムのモジュール化が進むと，必然的に多数のソース・ファイルを管理しなければならなくなり，アセンブル/コンパイルやリンクなどの処理手順も複雑になってきます．

MAKEユーティリティは，このように複数のソース・ファイルに分けられたモジュールのアセンブル/コンパイル(分割コンパイル)を行う際に強力なツールとなります．

例として，プログラムprog.exeが，図1-2のようにC言語のソース・ファイルprog.cとsub.c，およびアセンブリ言語で記述されたasmsub.asmによって構成される場合を考えてみます．このとき，prog.cとsub.cでは，ヘッダ・ファイルhead.hを#include文を用いて取り込んでいるものとします．

〔表1-11〕MAKEの起動オプション

オプション	機能
/D	ファイル検索の際に，それぞれのファイルのタイム・スタンプ(日時)を表示する
/I	呼び出されたプログラムの終了コードを無視する．これによって，そのプログラムでエラーが発生しても，MAKEの処理は続行される
/N	MAKEが実行すべきプログラムを表示するだけで実行しない．MAKEファイルのデバッグに使用
/S	プログラム実行の表示をしない

〔図1-2〕分割コンパイルの例

〔リスト1-1〕 progmake.bat

```
1: cl -Zi -Fa -DLINT_ARGS -J -c prog.c
2: cl -Zi -Fa -DLINT_ARGS -J -c sub.c
3: masm /ZI /Z /ML asmsub,,asmsub;
4: link /CO /NOI /MAP /LI prog sub asmsub,,prog;
```

〔リスト1-2〕
プログラム prog.exe 生成用の MAKE ファイル

```
               ┌ターゲット      ┌ソース
記  ┌ 1: prog.obj: prog.c head.h
述  │ 2:      cl -Zi -Fa -DLINT_ARGS -J -c prog.c ←── コマンド
ブ  │ 3:     └──スペースまたはタブ
ロ  └
ッ    4: sub.obj: sub.c head.h
ク    5:     cl -Zi -Fa -DLINT_ARGS -J -c sub.c -c
      6:
      7: asmsub.obj: asmsub.asm
      8:     masm /ZI /Z /ML asmsub,,asmsub;
      9:
     10: prog.exe: prog.obj sub.obj asmsub.obj
     11:     link /CO /NOI /MAP /LI prog sub asmsub,,prog;
```

同図のプログラム prog.exe を作成するためのアセンブル/コンパイルは，**リスト1-1** に示す BAT ファイルを用いてバッチ処理を行うことにより自動化することが可能です(同リストの行番号は，清書ユーティリティによって付加されたもの．以下，本書のソース・ファイルでは同様の行番号が付加されている)．

しかし，バッチ処理では，同図のうちのどれか一つだけのソース・ファイルが更新された場合に，すべてのソース・ファイルに対して再アセンブル/再コンパイル作業が行われ，ソース・ファイルの数が多くなると，その再アセンブル/再コンパイルによる無駄な時間は無視できないものになってしまいます．

一方，MAKE ユーティリティでは，**リスト1-2** に示すようなモジュールやソース・ファイルの依存関係を指定した MAKE ファイルを指定することによって，必要とする再アセンブル/再コンパイルのみが自動的に行われます．たとえば，ソース・ファイル asmsub.asm を更新すれば，そのファイルだけのアセンブルとリンク処理だけが選択的に行われます．また，たとえばヘッダ・ファイル head.h を更新したとすれば，prog.c と sub.c が再コンパイルされ，それにともなってリンク処理も自動的に行われます．

ここで MAKE ファイルの一般的な構文は，次のようになります．

MAKE ファイルの構文

```
[macrodefinition]
        ⋮
outfile:infile [,…] [#comment]
[#comment]
    command [#comment]
    [command] [#comment]
        ⋮
```

macrodefinition では，一つ以上の MAKE マクロ定義を行うことができます．MAKE マクロ定義については後述することにします．

outfile は，ターゲットとなるファイル名であり，自動的に更新させたいファイル名を指定します．

infile には，outfile が依存するファイル名を指定します．たとえば，outfile が".exe"ファイルの場合は，infile は".obj"ファイルになります．また，outfile が".obj"ファイルの場合は，infile はソース・ファイル(".asm"や".c"など)になります．outfile と infile は"："(コロン)で区切ります．

command には，MS-DOS の外部コマンドを指定します．ここには，アセンブル/コンパイルやリンク処理などの手順を記述します．command 行は，何行でも記述することができますが，新しい行の先頭には，必ず一つ以上のタブ(TAB)かスペースを入れなければなりません．

comment フィールドは，"#"に続いてコメントを記述するために用いられます．

outfile フィールドから command フィールドの間を記述ブロックと呼びます．記述ブロックは，いくつでも指定することができますが，各記述ブロックの間には1行の空白行を入れなければなりません．

MAKE ユーティリティは，MAKE ファイルの内容を上から順番に処理していきます．したがって，処理指定(記述)の順番が重要になってきます．また，command の処理中にエラーが発生すると，その時点で MAKE ファイルの処理を中断します．

MAKE ファイルでは，マクロ定義も使用することができます．たとえば，**図1-2** の例でメイン処理を含むソース・ファイルが prog.c だけではなく，他のモジュールをリンクして sub.c モジュールや asmsub.asm モジュールは共用する必要があるとします．また，ライ

〔リスト1-3〕MAKEマクロの使用例
(macro.mak)

```
 1: TRG = prog  ←――マクロの定義
 2: LIB = slibc.lib
 3: OBJ = $(TRG).obj sub.obj asmsub.obj
 4: LNK = $(TRG) sub asmsub
 5:        ↙―――――――――↙――マクロの使用
 6: $(TRG).obj: $(TRG).c head.h
 7:      cl -Zi -Fa -DLINT_ARGS -J -c $(TRG).c
 8:
 9: sub.obj: sub.c head.h
10:      cl -Zi -Fa -DLINT_ARGS -J -c sub.c -c
11:
12: asmsub.obj: asmsub.asm
13:      masm /ZI /Z /ML asmsub, ,asmsub;
14:
15: $(TRG).exe: $(OBJ)
16:      link /CO /NOI /MAP /LI $(LNK), , $(TRG), $(LIB);
```

ブラリに関しても，ユーザがMAKEユーティリティを起動する時点で指定したい場合もしばしば生じます．このような場合には，MAKEマクロを用いることによってMAKEファイルを共用することができ，そのつど内容の一部異なるMAKEファイルを用意する必要がなくなります．

リスト1-3は，MAKEマクロを用いたMAKEファイルの一例です．ここで，ターゲット・ファイルのベース名となるマクロTRGや，ライブラリのベース名であるマクロLIBに対して必要なファイル名を与えるには次のように指定します．

make TRG＝prog1 LIB＝user macro.mak

これによって，MAKEマクロTRGにはprog1が指定され，MAKEマクロLIBにはuserが指定されます．したがって，最終的な実行ファイルはprog1.exeとなり，リンクの際にはuser.libがリンクされることになります．

＊　　　＊

この章では，まずMS-DOSの生い立ちから成長過程，および開発ユーティリティの利用方法について解説してきました．

MS-DOSは，当初CP/Mコンパチブルを謳い文句に登場しましたが，まもなくCP/M路線を見限り，UNIX指向へと大きな転換を遂げました．結果的には，このver.2.11における大変革が成功して，今日のソフトウェア・バスとして不動の地位を確立したといえます．そして今，ver.4.Xないしver.5.Xにおいてマルチタスク版への脱皮がささやかれています．

我々ユーザとしては，マイクロソフト社のソフトウェア開発力に期待し，MS-DOSマルチタスクの夢を近い将来において現実のものとして使いこなしてみたいものです．それが現実になったとしても，MS-DOS上のプログラム開発手順や開発ユーティリティの使用方法に関しては，大きな相違は考えにくいものがあります．したがって，われわれユーザは現在のユーティリティの利用方法を完全にマスタし，プログラムのより効率的な開発ができるようにしておくべきです．

とくに最近では，“Turbo××”などの例にみられるように，エディット/コンパイル/リンクなどの処理が，その言語プロセッサ内で簡単(自動的)にできてしまいます．この便利な機能ばかりを使っていると，あるときはコンパイル/リンク・オプションを変えてみたり，あるときは分割アセンブル/コンパイルを行うといった，本来のアセンブル/コンパイル/リンク手順を必要とする場合に戸惑うことになります．

そのような意味で，MS-DOS本来のプログラム開発手順やオプション類については，よく理解しておかなければなりません．そして，これらの手順や利用方法をすべて理解したうえで，エディタや言語プロセッサの便利な機能を使っていくべきではないでしょうか．

第2章 マクロ・アセンブラ MASM

セグメントとディレクティブとマクロ

この章では，MS-DOS上のアセンブリ・プログラムの開発において必須となる，マクロ・アセンブラMASMの機能，とくに疑似命令の使いかたについて解説します．

2-1 MASMの特徴

マクロ・アセンブラMASMは，8086 CPUシリーズ用の2パスのセルフ・アセンブラで，非常に強力で豊富な機能を備えています．その特徴をあげると次のようになります．

(1) 強力なマクロ機能
(2) 強力な条件アセンブル
(3) 8087シリーズ演算プロセッサ(コプロセッサ)のサポート
(4) 構造化(ストラクチャ)データのサポート
(5) 各種の豊富な疑似命令

このほかにMASM ver.5.1では，以下のような新しい特徴が追加されています．

(1) 80386 CPUと80387コプロセッサのすべての命令とアドレッシング・モードのサポート
(2) オブジェクト・ファイルに対して，デバッガCodeView用のデバッグ情報を埋め込むことができ，アセンブリ言語ファイルのソース・レベルでのデバッグが可能
(3) このバージョンから新しくサポートされた簡易セグメントの定義や，高級言語インターフェースが簡単に実現できる疑似命令の機能を追加
(4) 実数変数を初期化するステートメントをIEEE形式に変更

MASMは，その機能があまりにも豊富なため，使いかたは複雑なものとなっています．MASMの機能をすべて解説するには紙数の関係もあり，また，ややもするとかえってわかりにくいものになってしまうため，ここでは基本的な演算子と疑似命令を重点に解説していくことにします．

2-2 8086 CPUのセグメント

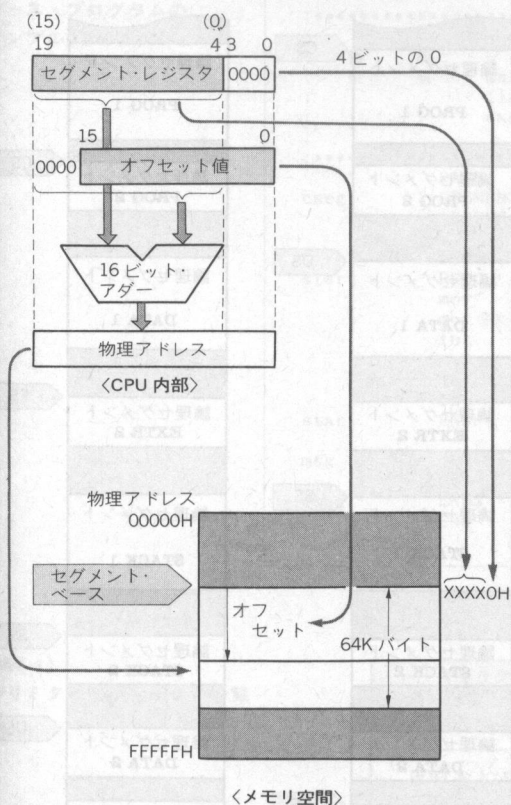
MS-DOSは，8086 CPU用のDOS(Disk Operating System)として開発されているため，メモリ管理などのDOSの仕様や，MS-DOSのプログラム開発ユーティリティなどは，8086 CPUのアーキテクチャの影響を強く受けたものとなっています．したがって，MS-DOS上でプログラムの開発を行う際には，第1章で解説したプログラム開発ユーティリティの機能をよく理解するとともに，8086 CPUのアーキテクチャについてもよく知っておかなければなりません．

セグメント方式のアドレッシング機構

8086 CPUは，8ビットCPUとの互換性を考慮しながら発展したCPUであるため，1Mバイトのアドレス空間をもっているにもかかわらず，IP(インストラクション・ポインタ)やSP(スタック・ポインタ)などのアドレッシングに関するレジスタ類を含むすべてのレジスタは16ビット長となっていて，それだけでは64Kバイトの空間しかアクセスすることができません．

そこで，8086 CPUではセグメントという概念を導入しています．これは，16ビットのセグメント・レジスタを4ビットだけ左にシフトして(16倍に相当する)，これにアドレッシング用のレジスタ値(オフセット)を加算することによって20ビットのアドレス値を得て，1Mバイトのアドレス空間をアクセス可能にするものです(図2-1)．

すなわち8086 CPUでは，まずセグメント・レジスタによってセグメント・ベースの指定を行います．ここで前述のような機構から，このセグメント・ベースは，1Mバイトのアドレス空間に対して16バイトごとに

〔図2-1〕 物理アドレスの生成とアドレス空間

指定することができます．

次に，このセグメント・ベースに対して16ビットのアドレス値を加えることによって，1バイトごとのアドレス指定を可能にしています．したがって，一つのセグメント・ベース(64Kバイト空間)は，16ビット・レジスタを使ってリニアにアクセスすることができ，あたかも8ビットCPUのアドレス空間と同様に扱うことが可能になっています．

セグメント方式の長所と短所

このセグメント導入の長所として，セグメント内(NEARという)のコールやジャンプ，あるいはデータへのアクセスが16ビットのオフセットのみで行えるため，従来の8ビットCPUと同様の考えかたでプログラミングすることができるという点があります．そして，これらの命令を機械語レベルで考えると，アドレス指定のオペランドに対するバイト数が最大でも2バイトとなるため，メモリの使用効率が向上することになります．

これに対して，セグメント導入の短所としては，一つのセグメントでは64Kバイトの空間しかアクセスできないため，それ以上のプログラムやデータをアクセスすることが必要な場合に，なんらかの方法でセグメント・レジスタの内容を変更しながら64Kバイト以上のコードやデータを扱えるようにしなければなりません．

そのために，プログラマは常にセグメントの管理を意識しながらプログラミングしなければならなくなります．また，アセンブラやリンカなどのプログラム開発ユーティリティでもセグメント管理に関する機能を追加しなければならなくなり，その機能がより複雑なものとなってしまいます．

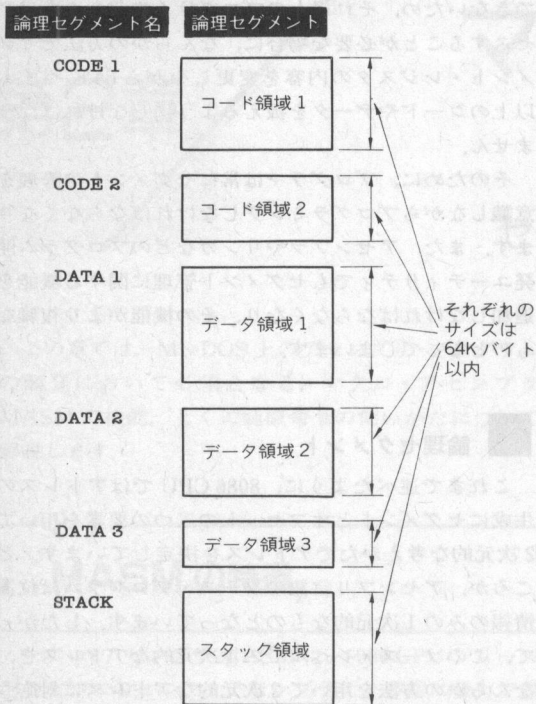
論理セグメント

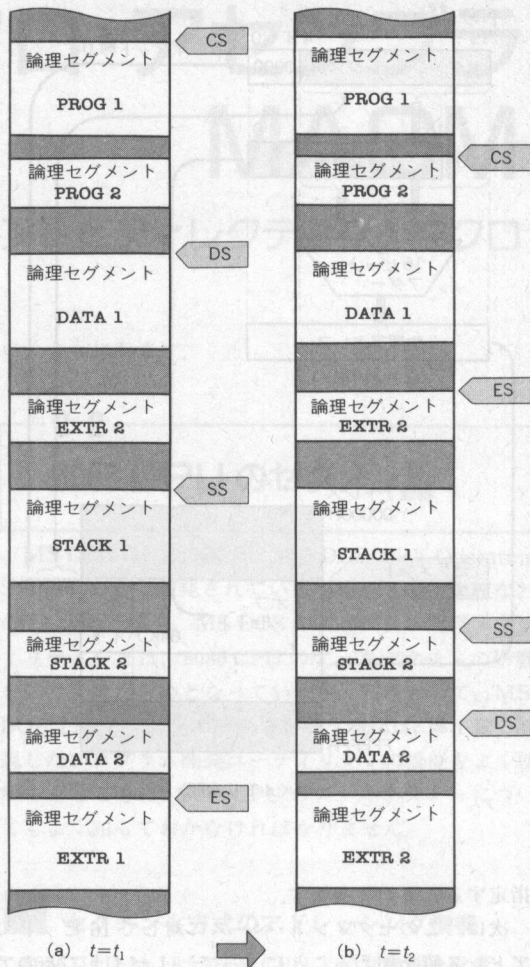
これまで述べたように，8086 CPUではアドレスの生成にセグメントとオフセットの二つの要素を用いて2次元的な考えかたでアドレスを決定しています．ところが，アセンブリ言語のソース・プログラムは位置情報のみの1次元的なものとなっています．したがって，このソース・レベルでの1次元的なアドレスを，なんらかの方法を用いて2次元的なアドレスに対応づける必要が生じてきます．

また，MS-DOSでは，そのプログラムがロードされる物理アドレスは，実際にそのプログラムがメモリ上にロードされるときまで決定できません．しかも，実行されるたびにそのロード・アドレス(セグメント値)が変わることもあるため，セグメント・アドレスをなんらかの方法で仮定してプログラムを作成する必要がでてきます．

そこでMASMでは，このセグメント仮定の問題を解決するために“論理セグメント”という概念を導入しています．論理セグメントとは，図2-2に示すように64Kバイト以内のある論理的なまとまりごとにプログラム(コード)領域，あるいはデータ領域を定義して参照できるように名前をつけたものです．これは，8086 CPUの実際のセグメント・レジスタが指す物理的なセグメント値とは区別して考えなければなりません．

この論理セグメントに対して，プログラム領域やデータ領域をどのようにまとめるかはプログラマに委ねられていて，プログラムの中で論理セグメントがいくつあってもかまいません．このように定義した論理セグメントを，MS-DOSがプログラム実行時に実際のセグメント・レジスタに割り振ることでセグメント・レジスタを時間的に変化させることができ，これによって64Kバイト以上のコードやデータにも対応することが可能になっています(図2-3)．

〔図2-2〕論理セグメント

CODE 1，CODE 2 の二つのコード領域と，DATA 1，DATA 2, DATA 3という三つのデータ領域と，STACKというスタック領域を，それぞれ論理セグメントとしてもつプログラムの例．

〔図2-3〕論理セグメントの割り当て

CS, DS, SS, ES の４個のセグメント・レジスタを，それぞれ論理セグメントに割り当ててプログラムの実行を進めていく．

2-3 アセンブリ言語の記述

アセンブリ言語でプログラムを記述する場合には，MASM によって定められている規則に基づいてソース・ファイルを作成します．

トークンとデリミタとセパレータ

MASM では，ソースとなる文(ステートメント)を記述する際に，行単位という制約はあるものの，自由欄形式のアセンブラであるため，ニーモニックのカラム指定などはなく，タブやスペースを自由に入れてステートメントを記述することができます．

リスト2-1 に示したtoken.asm は，MASM のソース・プログラムの記述例です．同リストにおいて，CODE や mov などのように意味のある最小の語句の単位を“トークン”といいます．また，トークンとトークンを切り離す役割をもつ語句(記号)を“セパレータ”といいます．セパレータには空白(20H)とタブ(09H)があり，これらを一般にホワイト・スペースと呼ぶこともあります．また，トークンやセパレータのほかに，１個のトークンの終わりを示す“デリミタ”と呼ばれるキャラクタがあります．同リスト中の，

，(カンマ)　；(セミコロン)　：(コロン)

などはすべてデリミタです．**表2-1** に示したのは，MASM における主なデリミタとセパレータの一覧です．

〔リスト2-1〕
ソース・プログラムのサンプル(その1)

```
;**************************************************************
;
;   機    能：ソース・プログラムのサンプル
;   生    成：masm /ML samp;
;             link /NOI/MAP samp;
;             exe2bin samp.exe samp.com;
;
;**************************************************************
        PAGE    , 132
cseg    SEGMENT BYTE PUBLIC 'CODE'  ;論理セグメント定義
        ASSUME  CS:cseg, DS:cseg

        ORG     100h                ;プログラム開始番地
start   PROC
        mov     dx, OFFSET msg
        mov     ah, 09h
        int     21h                 ;文字列表示
        mov     ah, 4Ch
        mov     al, 00h
        int     21h                 ;プログラム終了
start   ENDP

msg     DB      'ソース・プログラムのサンプル'
        DB      0Dh, 0Ah, '$'
cseg    ENDS
        END     start
```


〔表2-1〕
デリミタとセパレータの一覧

キャラクタ	機能
20H または 09H	トークンの分離(セパレータ) (20H：スペース，09H：TAB コード)
0DH および 0AH	ステートメントの終わり (0DH：改行コード，0AH：ライン・フィード)
,	複数のオペランドの区切り
''または""	文字列定数の開始と終了
( )	式の演算順位の指定
< >	ストラクチャ，レコードに対する初期化データ マクロ演算子
[ ]	レジスタ間接アドレッシング・モードのアドレス式
;	コメント・フィールドの開始
:	NEAR ラベルの生成 セグメント・オーバライド演算子 EXTRN ディレクティブの型指定 ASSUME ディレクティブのセグメント指定 レコード・フィールドの定義 PROC ディレクティブにおける引数の型指定
.	ストラクチャ・フィールドの参照
$	現在のロケーション・カウンタ
=	数値等価記号(ディレクティブ)，再定義可能
+	加算演算子または符号
−	減算演算子または符号
*	乗算演算子
/	除算演算子
?	変数の宣言で初期化を指定しない(ゼロに初期化)
@	簡略化セグメントにおけるセグメント等価記号の先頭
_	名前に使用できるキャラクタ
& ! %	マクロ演算子(表2-6参照)

名　前

MASMでは，実際の数値やアドレスを直接書くだけでなく，その値やアドレスに対して“名前”をつけることができます．

プログラムの中で意味のある値(アドレス)に名前をつけておけば，その名前で値を参照することが可能となるため，プログラムの開発効率や保守性が格段に向上することになります．

名前の表す値としては，アドレスやデータや定数などがあります．これらの名前に使用できる文字は，

A~Z　a~z　0~9　?　_　$

です．ただし，名前の最初の文字として数字を用いることはできません．

名前の字数は何文字でもかまいませんが，先頭から31文字までしかチェックされないので，これ以上長い部分は無効になり，字数のみが有効となります．また，MASMの起動時に/MXまたは/MLオプションが指定されない場合，名前の大文字と小文字は区別されません．

ステートメント

MASMは，行単位で書式が自由なアセンブラであり，MASMはその行のトークンやデリミタを分解してプログラムとして解釈します．

このアセンブルの対象となるソース・プログラムの1行を“ステートメント”と呼びます．ステートメントのフォーマット(型式)は，次に示すように最大4個のフィールドを使用して記述し，1行のステートメン

● コメントのすすめ ●

筆者もプログラムの記述に関しては自信がありません．なぜなら，自分で記述したプログラムをあとで見直してみると，他人の作ったプログラムではないかと思うくらい「別モノ」に見えるからです．プログラムは自分が生んだいわば「分身」なのですが，時間とともに「アカの他人」になってしまうのですから，なんとも情けない限りです．

自分で書いたプログラムの内容が後になってわからなくなるのはコメントが不足しているからです．自分で書いたプログラムを自分で理解できないのですから，他の人が理解できるはずがありません．最近では，この問題に気がつき，できるだけ詳しいコメントを入れるように努力しています．

幸いにして，今日では優れた日本語FEPも出回り，コメントにも日本語が入れやすくなってきました．とくに筆者のように英語不精のプログラマにとってはありがたい存在といえます．

しばしば，商品としてのソース・リストを見る機会がありますが，プログラム本体部分の2倍以上ものスペースを，コメントのために使っていることが珍しくありません．

人に読まれる(あるいは自分で読み返す)プログラムを記述するには，プログラムの構造化とともに，最低でもつぎの項目はコメントとして入れておくべきでしょう．

① そのルーチンの機能

そのルーチンのもつ機能を詳しくコメントしておきます．これによって，プログラムを読む場合に，プログラムの1行1行を丁寧に読む必要がなくなります．

② 入出力パラメータ

入出力パラメータについて，その型や属性などを詳しくコメントしておきます．これと，①の機能の記述によって，そのルーチンをまったくのブラック・ボックス化して考えることができます．

③ アクセスしているグローバル変数

バグが発生した場合，よく経験することはグローバル変数への不当なアクセスです．筆者は，グローバル変数の最初の文字を大文字にするなどして，できる限りローカル変数と区別して記述するようにしています．

そして，できれば，

④ そのルーチンをアクセスしているルーチン名

⑤ そのルーチンからアクセスしているルーチン名

もコメントしておきます．これらのコメントもバグ退治には大きな威力を発揮します．

〔リスト2-2〕ソース・プログラムのサンプル(その2)

```
;*****************************************************************
;   機    能: ソース・プログラムのサンプル
;              (大文字→小文字変換プロシージャ)
;   生    成:   masm /ML source;
;
;*****************************************************************
          PAGE     , 132
          PUBLIC   tolower              ;外部参照
cseg      SEGMENT  BYTE PUBLIC          ;論理セグメント定義
          ASSUME   CS:cseg              ;論理セグメントの割り当て

tolower   PROC                          ;大文字→小文字変換プロシージャ
          cmp      al, 'A'              ;Aより小さい?
          jb       no_upper             ;英大文字ではないのでブランチ
          cmp      al, 'Z'              ;Zより大きい?
          ja       no_upper             ;英大文字ではないのでブランチ
          add      al, 'a' -'A'         ;大文字に大文字と小文字のコード差(20H)を加算
no_upper:
          ret                           ;サブルーチン・リターン
tolower   ENDP
cseg      ENDS                          ;論理セグメント終了
          END
name      operation operand             comment フィールド
フィールド フィールド フィールド
```

トは128文字以内でなければなりません．また，複数行にまたがるステートメントは許されません．

[name] [operation] [operand] [;comment]

リスト2-2(source.asm)は，ソース・プログラムの一例です．これらのフィールドはオプショナルで，命令によっては省略可能な場合もあり，また，ある種の命令では必須となるフィールドもあります．

● **name フィールド**

名前を記述するフィールドです．この名前は，そのステートメントをほかのステートメントから名前でアクセス可能にするためのラベル名となります．

● **operation フィールド**

ステートメントの動作を指定するフィールドで，このフィールドはCPU命令であるニーモニック，MASMの命令である疑似命令(ディレクティブ)，マクロ名やストラクチャ名，レコード名などを記述します．

● **operand フィールド**

ステートメントの動作対象となるデータを定義するフィールドで，なんらかの式や疑似命令にともなう引数などを記述します．

● **comment フィールド**

セパレータ“;”が現われると，それ以降の行末までがコメントとして認識されます．コメントは，MASMに対しては何の影響も与えないので，命令そのものに反映されることはありません．しかし，プログラム保守のうえでは，できるだけ詳しいコメントの記述を心がけるべきです．

2-4 ディレクティブ(疑似命令)

ディレクティブ(指示語)とは，MASM自体の機能を制御する命令(疑似命令)であり，オブジェクト・コードとしてメモリ内に直接展開されるようなことはありません．**表2-2**にMASM ver.5.1のディレクティブの一覧を示します．

MASMには，同表のようにディレクティブが豊富に備えられており，非常に強力なマクロ機能や条件アセンブルを実現しています．これらのディレクティブやマクロ機能，および条件アセンブルを駆使すれば高度なテクニックによるプログラミングも可能になっています．

しかし，これらのディレクティブやマクロ/条件アセンブルの機能をすべて理解して使いこなすのは容易ではありません．ここでは，これらのディレクティブのうち，特に使用頻度の高いものや，基本的なディレクティブをピックアップして解説していくことにします．

〔表2-2〕MASMのディレクティブ一覧 ①

分　類	ディレクティブおよび構文	機　能
簡略化 セグメント	.MODEL memorymodel [,language]	プログラムのメモリ・モデルを指定し，名前の呼び出し方法や引数の扱いかたを，どの言語に合わせるかをlanguageに指定する．memorymodelはSMALL/COMPACT/MEDIUM/LARGE/HUGEのいずれかを指定する
	.CODE [name]	.MODELディレクティブを使用している場合に，コード・セグメントの始まりを指示する．memorymodelがMEDIUM/LARGE/HUGEの場合はnameにセグメント名を指定できる
	.DATA	.MODELディレクティブを使用している場合に，初期化済みnearデータ・セグメント(_DATA)の開始を指示する
	.DATA?	.MODELディレクティブを使用している場合に，未初期化nearデータ・セグメント(_BSS)の開始を指示する
	.FARDATA [name]	.MODELディレクティブを使用している場合に，初期化済みfarデータ・セグメント(FAR_DATAまたはname)の開始を指示する
	.FARDATA? [name]	.MODELディレクティブを使用している場合に，未初期化farデータ・セグメント(FAR_BSSまたはname)の開始を指示する
	.CONST	.MODELディレクティブを使用している場合に，定数データ・セグメント(CONST)の開始を指示する
	.STACK [size]	.MODELディレクティブを使用している場合に，スタック・セグメント(STACK)の開始を指示する．sizeはスタックのバイト数(デフォルト1024)
データ・ アロケーション	name DB intializer [,…]	各initializerに1バイトの領域を割り当てて初期化する
	name DW intializer [,…]	各initializerに1ワード(2バイト)の領域を割り当てて初期化する
	name DD intializer [,…]	各initializerにダブル・ワード(4バイト)の領域を割り当てて初期化する
	name DF intializer [,…]	各initializerに1farワード(6バイト)の領域を割り当てて初期化する
	name DQ intializer [,…]	各initializerに1クワッド・ワード(8バイト)の領域を割り当てて初期化する
	name DT intializer [,…]	各initializerに10バイトの領域を割り当てて初期化する
セグメント	name SEGMENT [align] [combine] [use] ['class'] ⋮ name ENDS	nameの名前で論理セグメントを定義する．alignにはBYTE, WORD, DWORD, PARA, PAGEがある．combineにはPUBLIC, STACK, COMMON, MEMORY, ATaddress, PRIVATEがある．USEには，USE16とUSE32がある．'class'は，セグメントの識別やセグメント順序の制御に使用される
	name GROUP segment [,…]	グループを構成するsegmentをまとめてnameによって参照可能とする
	ASSUME segmentregister : name [,…] または ASSUME segmentregister : NOTHING または ASSUME NOTHING	各segmentregisterが，論理セグメントのどのnameに対応しているのかをMASMに指示する．NOTHINGは以前のASSUMEディレクティブによって指示されたsegmentregisterへの割り当てのうち，一部または全部の割り当てを取り消す
	DOSSEG	MS-DOSのセグメント順序に関する規約にしたがってセグメントを順序づける
	END [startaddress]	ソース・プログラムの終了を指示する．startaddressには，プログラムの開始アドレスを指定する
	.ALPHA	セグメントをアルファベット順に並べる
	.SEQ	セグメントを表れた順序に並べる

〔表2-2〕MASMのディレクティブ一覧 ②

分類	ディレクティブおよび構文	機能
コード/ラベル	label PROC [NEAR|FAR] [USES [reglist] ,] [argument [,…]] ⋮ label ENDP	このプロシージャ(サブルーチン)がNEAR CALLされるのかFAR CALLされるのかを宣言する. labelはプロシージャの先頭を指すラベルとして使用される. reglistはプロシージャが使用するために退避すべきレジスタのリストで, 空白またはタブで区切る. argumentはプロシージャにスタックで渡される引数
	name LABEL distance	命令コードを含むロケーションを参照するためのラベルを定義する. nameはラベルに割り当てるシンボル名であり, distanceにはnameがラベル名の場合にNEAR/FAR, 変数名の場合にBYTE/WORD/DWORD/QWORD/TBYTEを指定する
	ALIGN number	numberの倍数に等しいロケーションにアラインを行う. もし, ロケーション・カウンタが指定される境界になければ, NOP命令を挿入してロケーション・カウンタを指定境界までインクリメントする
	EVEN	常に次の偶数バイト(ワード境界)に対してアラインを行う
	ORG expression	論理セグメントのロケーション・カウンタにexpressionの値を代入する. これによって, このあとのコードおよびデータはexpressionによって指定される新しいオフセットから始まる
有効範囲	PUBLIC name [,…]	nameとして指定した各変数, ラベル, または絶対シンボルを, 他のモジュールから参照可能にする
	EXTRN name : type [,…]	他のモジュールでPUBLIC宣言されたnameをtype型として定義して, そのモジュール内で使用可能にする
	COMM definition [,…]	definitionで指定した共有変数を宣言する. definitionの構文は, [NEAR|FAR] label : size [: count] labelは変数の名前であり, sizeには型指定子(BYTE/WORDなど)を指定する. countはデータの個数でありデフォルトは1
	INCLUDELIB library	そのモジュールをlibraryとリンクするようにリンカLINKに対して指示する
	LOCAL verdef	スタック上にローカル変数を割り当てる. verdefの構文は, variable[[count] [[NEAR|FAR] PTR] type]] variableはローカル変数の名前であり, typeは割り当てる変数の型である. countは指定した名前と型の要素をスタック上に割り当てる数であり, 角カッコで囲む
構造体とレコード	recordname RECORD field [,…]	指定したfieldからなるrecordnameと型を宣言する. fieldの構文は, fieldname : width [=expression] fieldnameはフィールドの名前であり, widthはビット数を指定する. expressionで初期値を与えることができる
	name STRUC ⋮ name ENDS	構造化したデータをnameで定義されたフィールド名を用いて一括してアクセス可能にする
マクロ	name MACRO [parameter [,…]] ⋮ ENDM	nameで呼び出されるマクロ・ブロックを定義し, parameterをマクロ呼び出し時に渡される引数の置換部分として作成する
	EXITM	現在の繰り返しブロックまたはマクロ・ブロックの展開を終了し, そのブロック外の次のステートメントのアセンブルを開始する
	LOCAL localname [,…]	マクロ定義内で使用するシンボルとしてlocalnameを宣言する

〔表2-2〕MASMのディレクティブ一覧 ③

分類	ディレクティブおよび構文	機能
マクロ	PURGE macroname [,…]	macronameで指定されたマクロ・ブロックをメモリから削除する
	textlabel CATSTR string [,…]	引数として与えられたstringを連結し，textlabelに代入する
	numericlabel INSTR [start,] string1, string2	string1の中の部分文字列string2の位置をnumericlabelに返す
	numericlabel SIZESTR string	stringの長さをnumericlabelに返す
	textlabel SUBSTR string, start [,length]	stringの中からstart，lengthで指定された部分文字列を取り出し，textlabelに返す
等価記号	name EQU [〈] expression [〉]	expressionをnameに割り当てる．expressionを山形カッコで囲むとテキストになる．EQUディレクティブで定義した数値等価記号は再定義できないが，テキスト等価記号は再定義可能
	name = expression	expressionの数値をnameに割り当てる．シンボルの再定義可能
リピート・ブロック	REPT expression : ENDM	このブロックのステートメントをexpressionで指定された回数だけ繰り返して展開する
	IRP parameter, 〈argument [,…]〉 : ENDM	parameterに山形カッコで囲まれたargumentを置き換えてマクロ展開を繰り返す
	IRPC parameter, string : ENDM	parameterにstring内の1文字ずつを置き換えてマクロ展開を繰り返す
条件アセンブル	IF expression :(ifstatements) [ELSE :(elsestatements)] ENDIF	expressionが真(非ゼロ)の場合ifstatementsをアセンブルする．もし，ELSEが指定されていれば，expressionが偽(ゼロ)の場合にelsestatementsをアセンブルする
	IF1	パス1のときだけアセンブルする
	IF2	パス2のときだけアセンブルする
	IFB 〈argument〉	argumentが空白の場合にアセンブルする
	IFDEF name	nameが以前に定義されている場合にアセンブルする
	IFDIF [I] 〈argument1〉, 〈argument2〉	argument1とargument2が不一致の場合にアセンブルする．Iを指定すると大文字/小文字の区別をしない
	IFE expression	expressionが偽(ゼロ)の場合にアセンブルする
	IFIDN [I] 〈argument1〉, 〈argument2〉	argument1とargument2が同一の場合にアセンブルする．Iを指定すると大文字/小文字の区別をしない
	IFNB 〈argument〉	argumentが空白でない場合にアセンブルする
	IFNDEF name	nameが定義されていない場合にアセンブルする
	ELSEIF …	条件が偽のときにアセンブルされる条件ブロックの始まりを示す
条件エラー	.ERR	エラーを発生する
	.ERR1	パス1のときだけエラーを発生する
	.ERR2	パス2のときだけエラーを発生する
	.ERRB 〈argument〉	argumentが空白のときエラーを発生する
	.ERRDEF name	nameが以前に定義されているときエラーを発生する
	.ERRDIF [I] 〈argument1〉, 〈argument2〉	argument1とargumest2が不一致の場合にエラーを発生する．Iを指定すると大文字/小文字の区別をしない
	.ERRE expression	expressionが偽(ゼロ)の場合にエラーを発生する

〔表2-2〕MASMのディレクティブ一覧 ④

分　類	ディレクティブおよび構文	機　能
条件エラー	.ERRIDN [I] 〈argument1〉, 〈argument2〉	argument1とargument2が一致する場合にエラーを発生する．Iを指定すると大文字/小文字の区別をしない
	.ERRNB 〈argument〉	argumentが空白でない場合にエラーを発生する
	.ERRNDEF name	nameが以前に定義されていない場合にエラーを発生する
	.ERRNZ expression	expressionが真(非ゼロ)の場合にエラーを発生する
プロセッサ	.8086	8086の命令をアセンブル可能にし上位プロセッサ用の命令を禁止する(デフォルト)
	.186	80186プロセッサ用の命令をアセンブル可能にする
	.286	80286プロセッサ用の非特権命令をアセンブル可能にする
	.286P	80286プロセッサ用の特権命令を含むすべての命令をアセンブル可能にする
	.386	80386プロセッサ用の非特権命令をアセンブル可能にする
	.386P	80386プロセッサ用の特権命令を含むすべての命令をアセンブル可能にする
	.8087	8087の命令をアセンブル可能にし，上位のコプロセッサ用の命令を禁止する(デフォルト)
	.287	80287コプロセッサの命令をアセンブル可能にする
	.387	80387コプロセッサの命令をアセンブル可能にする
リスト出力の制御	TITLE text	リストの各ページの最初の行にtextで指定されたタイトルを出力する
	SUBTTL text	リストの各ページのタイトルの次の行にtextで指定されたサブタイトルを出力する
	PAGE [[length] , width]	リスティング・フォーマットをlength行，width幅に設定する
	PAGE +	ページ番号に1を加える
	.LIST	すべての行を命令コードとともに出力する(デフォルト)
	.XLIST	この後の行の出力を禁止する
	.LFCOND	偽の条件ブロックの行をリスト出力する(デフォルト)
	.SFCOND	偽の条件ブロックの行をリスト出力しない
	.TFCOND	.SFCONDや.LFCONDの条件を反転する
	.XALL	マクロ展開の結果，コードまたはデータを生成する行を出力する(デフォルト)
	.LALL	すべてのマクロを展開し，その行をすべて出力する
	.SALL	マクロ展開のリスト出力を禁止する
	.CREF	クロス・リファレンス・ファイルを作成する(デフォルト)
	.XCREF [name [,…]]	クロス・リファレンス・ファイルに対してシンボルの出力を禁止する．nameを指定すると指定したシンボルだけが禁止される
その他	COMMENT delimiter [text] text delimiter [text]	COMMENTディレクティブの後の最初の空白以外の文字をdelimiterとして，次のdelimiterに出会うまでのtextをコメント・ブロックとして扱う
	%OUT text	textを標準出力(スクリーン)に出力する
	.RADIX expression	数値を読み込む際の基数をexpressionで設定する
	END [startaddress]	プログラム・モジュールの終わりを示す．startaddressはオプションであり，プログラムの開始番地を指定する
	INCLUDE filespec	filespecで指定したソース・ファイルをアセンブル中のソース・ファイルに挿入する
	NAME modulename	ver.5.1では無視される．モジュール名は常にソース・ファイル名のベース名となる

モジュール・プログラミング

あるプログラムを作成する場合に，そのプログラムの機能を細分化/分割してモジュールとして作成し，最終的にそれらの細分化されたプログラム群を組み合わせて一つの機能を実現するプログラムの作成方法を"モジュール・プログラミング"と呼んでいます．

MS-DOSでは図2-4に示したように，このモジュール・プログラミングを促進するために，一つのプログラムは一つ以上のモジュール(個々のソース・プログラムから作成されたオブジェクト・プログラム)を集めてリンクして構成するという方法を採用しています．

また，個々のソース・プログラムをさらに細分化していけば，たとえば入出力処理やワーク・エリア，定数エリアなど，コードやデータを論理的にまとめた論理セグメントに分けることができます．

論理セグメントは，さらにサブルーチンをブロック化したプロシージャやデータなどに分けることができます．MASMでも，プログラムの構造化を図るためにモジュールや論理セグメント，プロシージャなどの定義に関して豊富なディレクティブが用意されています．

● モジュール名

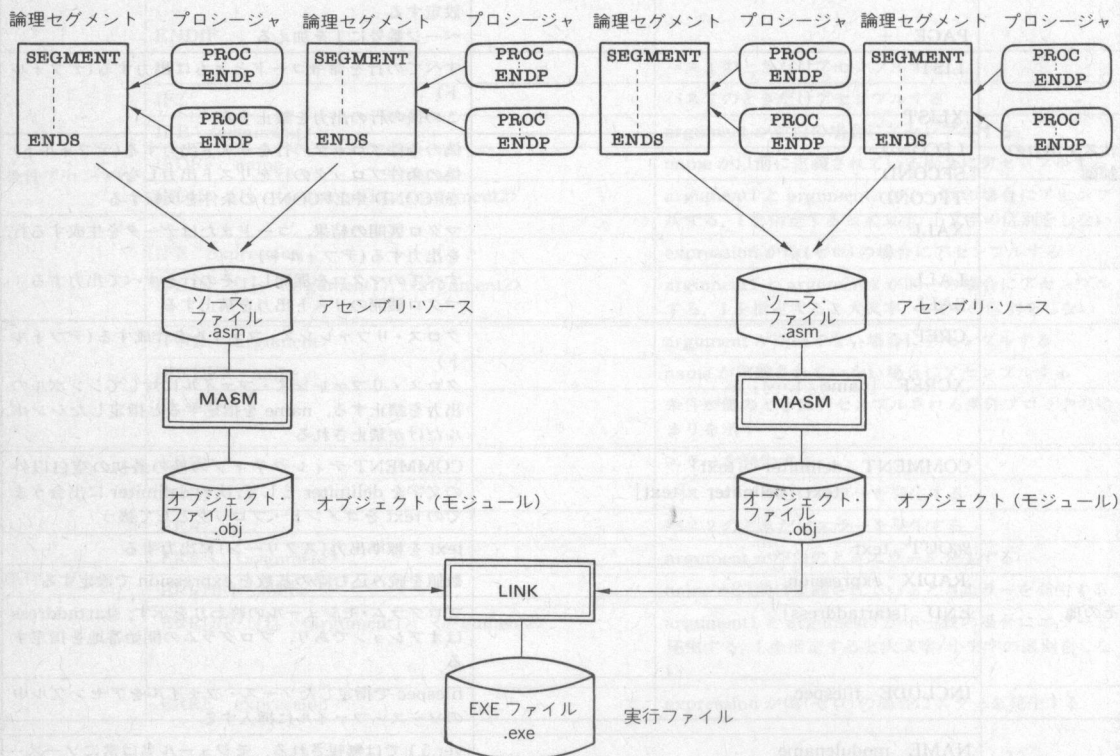
MS-DOSでは，1本のソース・プログラムからアセンブル/コンパイルされた個々のオブジェクト・ファイルの単位を"モジュール"と呼んでいます．各モジュールには，その中にモジュール名が設定され，そのモジュール名はLINKに渡されます．

MASMの従来のバージョンでは，モジュール名の指定を行うためにNAMEディレクティブが用意されていました．

NAME modulename

このNAMEディレクティブは，モジュール名の定義を行います．ここでmodulenameは，予約語以外の名前として有効な文字列です．

MASM ver.5.1では，ソース・プログラム・ファイルのベース名(拡張子を除いた部分)をモジュール名として自動的にオブジェクト・ファイルに書き込みます．このためver.5.1では，このNAMEディレクティブは(以前のバージョンと互換性を保つために認識はされるが)何の効力ももちません．すなわち，このNAMEディレクティブに対してmodulenameを指定しても，そのmodulenameは無視されます．

〔図2-4〕モジュール・プログラミング

● セグメントの定義

MASMでは，命令コードやデータを1個あるいは複数の論理セグメントと呼ばれるブロックの中に配置させなければなりません．

この論理セグメントを定義するのが，SEGMENTとENDSディレクティブです．ただし，ここでのセグメントとは，CPUのセグメント・レジスタの示す物理的なセグメントとは異なり，あくまでもMASMやLINKで扱う論理的なセグメントであることに注意しなければなりません．

すなわち，CPUのCS(コード・セグメント)レジスタやDS(データ・セグメント)レジスタがどのアドレスを指していようとも，ユーザが指定したとおりにリンクされていくので，この点を考慮してセグメントを記述しなければなりません．これらのセグメント情報は，のちにLINKや，ひいてはプログラムをロードする際のリロケート情報としてMS-DOSに渡されます．

論理セグメントは，データ領域や命令コード領域を分離したり，逆に命令やデータの領域を同じセグメントに配置したりします．後者の方法では，このほかに外部モジュール内で記述されているサブルーチンや関数を参照しないで，64Kバイトに収まる限り(COMモデル)，8ビットCPUのアセンブリ言語と大差のないコーディングが可能となります．

しかし，8086 CPUのアーキテクチャを最大限に有効に発揮しようとするならば，前者のように論理セグメントを分離して考えたほうが，プログラムの保守性の点からみてもメリットがあります．

SEGMENT ENDSの構文

```
name SEGMENT[align] [combine] [use] ['class']
   :
name ENDS
```

SEGMENTディレクティブでは，セグメントのname以外にalign, combine, use, classという四つの属性をもち，これらの属性によりLINKに対して論理セグメントのメモリ配置に関する指定を行います．

◆ nameの指定

nameはセグメントの名前であって，省略することはできません．MASMは，この同一のnameが付いているSEGMENTとENDSディレクティブで囲まれたブロックを一つの論理セグメントとみなします．また，もし複数の論理セグメントに対して同じnameをつけた場合は，それらは同一のセグメントとして扱われます．

◆ alignの指定

alignは，論理セグメントのセグメント・ベース(始まり)を物理的なメモリ配置に関して指定するもので，BYTE/WORD/DWORD/PARA/PAGEの五つのタイプがあります(図2-5)．この情報はLINKに渡され，プログラムのロード時にセグメントの開始アドレスが指定された配置でリロケートされます．

▶ BYTE

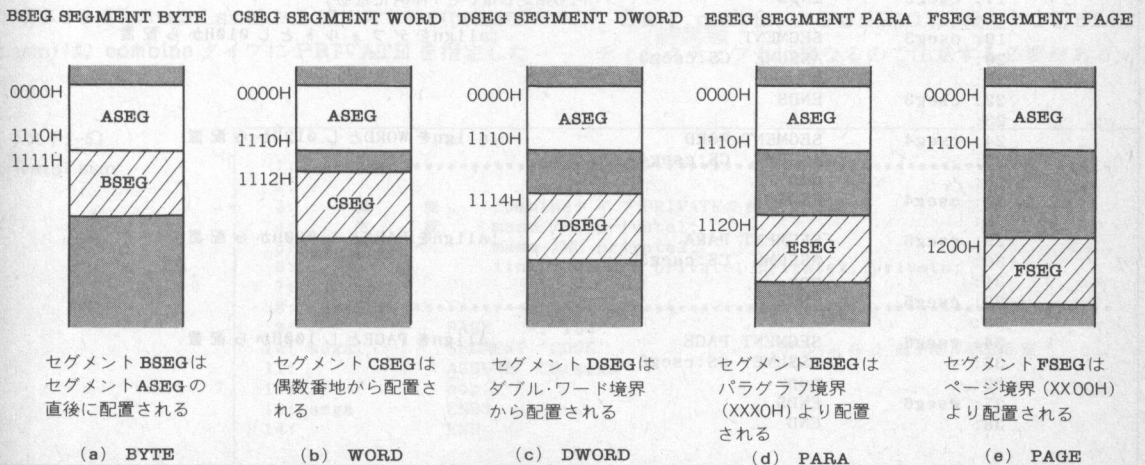
バイト単位でセグメントを配置するので，あたかもプログラムやデータが一つのセグメントで展開されているかのごとくにリンクされる．したがって余分なスペースを必要としないのでメモリの利用効率が良い．

▶ WORD

アドレスの下位1ビットを無視し，メモリの偶数番地からセグメントを配置する．

▶ DWORD

ダブル・ワード・アドレスに対してセグメントを配置する．これは，80386 CPUに対応して用意されたもので，80386 CPUの32ビット・セグメントでは，通常のalignとしてこのDWORDが使用される．

〔図2-5〕各alignのタイプによるセグメント配置の相違

▶ **PARA**

パラグラフ(上位16ビット)の先頭から配置する．alignに何も指定しない場合は，このPARAが選択される(デフォルト)．

▶ **PAGE**

メモリの下位8ビットが0，すなわち256バイト単位でセグメントが開始される．

【alignのサンプル・プログラム】

リスト2-3(align.asm)は，これらのalignの各タイプの指定によるメモリ配置の違いを示すサンプル・プログラムです．なお，以降のソース・リストでは，清書ユーティリティによって行番号が付加され，リストを読む際の一助としています．

同リストのプログラムをアセンブル/リンクすることにより，**リスト2-4**(align.map)に示すMAPファイルを得ることができ，そのメモリ配置は図2-6のように表すことができます．

cseg1ではalignタイプにBYTEが指定され，開始アドレスである00000Hから配置されます．cseg2もBYTEで指定されているため，前のcseg1に引き続き00001Hから配置されます．

次に，cseg2は00001Hで終わっていますが，cseg3では何も指定していないので，デフォルトのPARAを指定したことになり00010Hから配置されます．

cseg3は00010Hで終わっていて，以後のセグメントは00011Hから配置可能ですが，cseg4に対してWORDを指定しているため，偶数番地すなわち00012Hから配置されます．

同様に，cseg5も00013Hから配置可能になっていますが，cseg5に対してPARAを指定しているため00020Hから，cseg6に対してはPAGEを指定したためにページ単位である00100Hから配置されています．

◆ **combineの指定**

combineは，そのセグメントが他のモジュールとリンクされ，実際にプログラムがロードされる際の，他のセグメントとの相対的な配置を指定するものであり，次に示すPRIVATE，PUBLIC，COMMON，STACK，ATの五つが用意されています．

▶ **PRIVATE**

combineの指定が省略された場合，その論理セグメントは，他のモジュールの論理セグメントや同一モジュール内のnameの異なる論理セグメントとは別のセ

〔リスト2-3〕 alignの指定(align.asm)

```
 1: ;****************************************************************
 2: ;
 3: ;    機    能:   アライン・タイプの違い
 4: ;    生    成:   masm /ML align;
 5: ;                link /NOI/MAP align,,align;
 6: ;
 7: ;****************************************************************
 8:                 PAGE    , 132
 9: cseg1           SEGMENT BYTE                  ;alignをBYTEとする
10:                 ASSUME  CS:cseg1
11:                 nop
12: cseg1           ENDS
13:
14: cseg2           SEGMENT BYTE                  ;alignをBYTEとし001Hから配置
15:                 ASSUME  CS:cseg2
16:                 nop
17: cseg2           ENDS
18:
19: cseg3           SEGMENT                       ;alignをデフォルトとし010Hから配置
20:                 ASSUME  CS:cseg3
21:                 nop
22: cseg3           ENDS
23:
24: cseg4           SEGMENT WORD                  ;alignをWORDとし012Hから配置
25:                 ASSUME  CS:cseg4
26:                 nop
27: cseg4           ENDS
28:
29: cseg5           SEGMENT PARA                  ;alignをPARAとし020Hから配置
30:                 ASSUME  CS:cseg5
31:                 nop
32: cseg5           ENDS
33:
34: cseg6           SEGMENT PAGE                  ;alignをPAGEとし100Hから配置
35:                 ASSUME  CS:cseg6
36:                 nop
37: cseg6           ENDS
38:                 END
```

〔リスト2-4〕
align 指定によるメモリ配置

```
LINK : warning L4021: no stack segment

 Start  Stop   Length Name                                    Class
 00000H 00000H 00001H cseg1 ←── BYTE (開始)
 00001H 00001H 00001H cseg2 ←── BYTE (奇数)
 00010H 00010H 00001H cseg3 ←── デフォルト (PARA : パラグラフ)
 00012H 00012H 00001H cseg4 ←── WORD (偶数)
 00020H 00020H 00001H cseg5 ←── PARA (パラグラフ)
 00100H 00100H 00001H cseg6 ←── PAGE (ページ)

  Address         Publics by Name

  Address         Publics by Value
```

〔図2-6〕 align のタイプ別におけるメモリ配置

グメントとして扱われ，そのモジュール内だけの独自の物理セグメントが与えられる．

【PRIVATE のサンプル・プログラム】

リスト2-5(private1.asm)および**リスト2-6**(private2.asm)は，combine タイプに PRIVATE を指定した例を表しています．同リストは，双方ともにセグメントの名前が sega であり，クラス名も CODE となってまったく同一のセグメントとして定義されていますが，combine タイプにデフォルトの PRIVATE が指定されています．

これをアセンブル/リンクすると**リスト2-7**(private.map)の MAP ファイルを得ることができます．同リストから，combine タイプとして PRIVATE が指定されていると，同じセグメント名，クラス名であっても，それぞれ独自のセグメントとして扱われていることがわかります(**図2-7**)．

ただし，これらのセグメントが同じファイル(モジュール)の中で定義された場合は，当然同じセグメントとして扱われるので注意が必要です．

▶ PUBLIC

異なるモジュールで定義された複数のセグメントが同じ name あるいは class をもつ場合，それらのセグメントは，リンク時にまとめて一つのセグメントになる．

そして，これらのセグメントは同じセグメントとして扱われるため，アドレスは共通のセグメント・ベースからのオフセットになる．

ここで，combine の PUBLIC は，後述の PUBLIC ディレクティブとは異なるので注意する必要がある．

〔リスト2-5〕
private1.asm

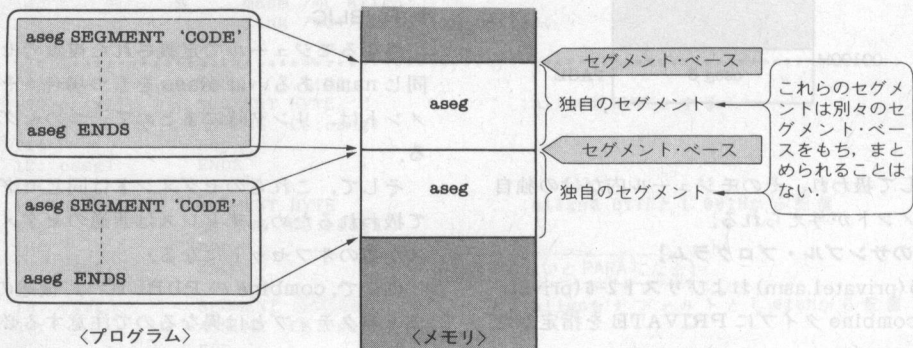
```
 1: ;*****************************************************************
 2: ;
 3: ;    機    能 :  combineタイプPRIVATEの例その1
 4: ;    生    成 :  masm /ML private1;
 5: ;                masm /ML private2;
 6: ;                link /NOI/MAP private1 private2,,private;
 7: ;
 8: ;*****************************************************************
 9:              PAGE     , 130
10: sega         SEGMENT 'CODE'          ;デフォルトのPRIVATE指定
11:              ASSUME  CS:sega
12:              nop
13: sega         ENDS
14:              END
```

〔リスト2-6〕 private2.asm

```
 1: ;**************************************************************************
 2: ;
 3: ;     機     能 :     combineタイプPRIVATEの例その2
 4: ;     生     成 :     masm /ML private1;
 5: ;                     masm /ML private2;
 6: ;                     link /NOI/MAP private1 private2,,private;
 7: ;
 8: ;**************************************************************************
 9:                 PAGE      130
10: sega            SEGMENT 'CODE'                  ;デフォルトのPRIVATE指定
11:                 ASSUME  CS:sega
12:                 nop
13: sega            ENDS
14:                 END
```

〔リスト2-7〕 combineタイプPRIVATEのMAPファイル

```
LINK : warning L4021: no stack segment

 Start  Stop   Length Name                  Class
 00000H 00000H 00001H sega                  CODE
 00010H 00010H 00001H sega                  CODE

  Address         Publics by Name


  Address         Publics by Value
```


〔図2-7〕 PRIVATE指定によるセグメント配置

【PUBLICのサンプル・プログラム】

リスト2-8(public1.asm)および**リスト2-9**(public2.asm)はcombineのPUBLICタイプを使用した例を示しています．

二つのリストにおいて，セグメントsegaは別々のモジュールで定義されていますが，combineタイプにPUBLICを指定し，セグメント名，クラス名ともに同一の名前を使用しているため，一つのセグメントにまとめられます．

一方，セグメントsegbは，combineタイプにPUBLICを指定し，クラス名もCODEになっていますが，セグメント名が異なるので別々のセグメントとして扱われます．

リスト2-10(public.map)は二つのモジュールをリンクした結果得られたMAPファイルです．同リストから，セグメントsegaは一つのセグメントにまとめられているものの，alignタイプにデフォルトのPARAが採用され，全体として2バイトのコードしか

〔リスト2-8〕public1.asm

```
 1: ;*************************************************************
 2: ;
 3: ;   機    能 :    combineタイプPUBLICの例その1
 4: ;   生    成 :    masm /ML public1;
 5: ;                 masm /ML public2;
 6: ;                 link /NOI/MAP public1 public2,,public;
 7: ;
 8: ;*************************************************************
 9:            PAGE    , 130
10: sega       SEGMENT PUBLIC 'CODE'                  ;PUBLICの指定
11:            ASSUME  CS:sega
12:            nop
13: sega       ENDS
14:
15: segb       SEGMENT PUBLIC 'CODE'                  ;PUBLICの指定
16:            ASSUME  CS:sega
17:            nop
18: segb       ENDS
19:            END
```

セグメント名が異なる

〔リスト2-9〕public2.asm

```
 1: ;*************************************************************
 2: ;
 3: ;   機    能 :    combineタイプPUBLICの例その2
 4: ;   生    成 :    masm /ML public1;
 5: ;                 masm /ML public2;
 6: ;                 link /NOI/MAP public1 public2,,public;
 7: ;
 8: ;*************************************************************
 9:            PAGE    , 130
10: sega       SEGMENT PUBLIC 'CODE'                  ;PUBLICの指定
11:            ASSUME  CS:sega
12:            nop
13: sega       ENDS
14:            END
```

〔リスト2-10〕combineタイプPUBLICによるMAPファイル

```
LINK : warning L4021: no stack segment

 Start  Stop   Length Name                   Class
 00000H 00010H 00011H sega                   CODE  ←— セグメントsegaはまとめて1個のセグメントになる
 00020H 00020H 00001H segb                   CODE  ←— セグメントsegbは別のセグメントとして扱われる

  Address         Publics by Name

  Address         Publics by Value
```

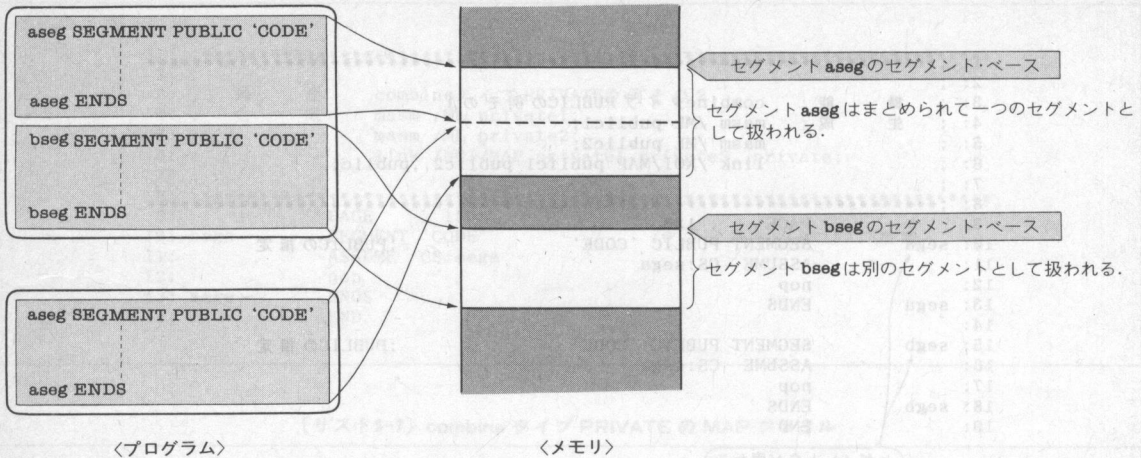
入っていないにもかかわらず，セグメントsegaの大きさが00011H(17バイト)になっています．

また，セグメントsegbにもalignにPARAが採用され，別のセグメントとして扱われるために，セグメント・ベースが00020Hになっています(図2-8)．

▶ **COMMON**

異なるモジュールで定義された複数のセグメントが同じnameあるいはclassをもつ場合，これらのセグメントは同一のアドレスからオーバラップ(重複)して同一のアドレスに配置される．この場合にセグメントの大きさは異なっていてもかまわない．

このcombine型は，たとえば一つの物理アドレスに対して複数の変数や定数を割り当てる場合などに使用される．

〔図2-8〕PUBLIC 指定によるセグメント配置

【COMMON のサンプル・プログラム】

リスト2-11(common1.asm)および**リスト2-12**(common2.asm)は，combine タイプに COMMON を使用した例を示しています．**リスト2-11**(common1.asm)において，セグメント dseg は二度に渡って定義され，それぞれ data1(1111H)と data2(2222H)の宣言を行っています．

これらは，combine タイプ PUBLIC によって定義されているために同一のセグメントとして扱われ，data1 と data2 が現われた順にオフセットづけ(メモリ割り当て)されます．

一方，**リスト2-12**(common2.asm)では，**リスト2-11**と同じクラス名およびセグメント名をもつセグメント dseg が，combine タイプ COMMON で定義され，やはり二度に渡って定義されていて，data3(3333H)，data4(4444H)および data5(5555H)が宣言されてい

〔リスト2-11〕
combine タイプ
COMMON の例
(その1)

```
 1: ;*********************************************************************
 2: ;
 3: ;   機    能 :   combineタイプ COMMONの 例 そ の 1
 4: ;   生    成 :   masm /ML common1;
 5: ;                masm /ML common2;
 6: ;                link /NOI/MAP common1 common2, common, common;
 7: ;
 8: ;*********************************************************************
 9:                PAGE     , 130
10:                EXTRN    data3:WORD, data4:WORD, data5:WORD
11: cseg           SEGMENT  PUBLIC 'CODE'               ;PUBLICの 指 定
12:                ASSUME   CS:cseg, DS:dseg
13:
14: start          PROC
15:                mov      ax, SEG data1
16:                mov      ds, ax                      ;DSレジスタの 設 定
17:                mov      bx, data1                   ;data1の 読 み 込 み
18:                mov      cx, data2                   ;data2の 読 み 込 み
19:                mov      dx, data3                   ;data3の 読 み 込 み
20:                mov      si, data4                   ;data4の 読 み 込 み
21:                mov      di, data5                   ;data5の 読 み 込 み
22:                mov      ah, 4Ch
23:                mov      al, 00h                     ;リターン・コード
24:                int      21h                         ;プログラム終了
25: start          ENDP
26: cseg           ENDS
27:
28: dseg           SEGMENT  PUBLIC 'DATA'               ;PUBLICの 指 定
29: data1          DW       1111h
30: dseg           ENDS
31:
32: dseg           SEGMENT  PUBLIC 'DATA'               ;PUBLICの 指 定
33: data2          DW       2222h
34: dseg           ENDS
35:                END      start
```

実際にデータを
読み込んで確認する

これらのセグメントは
1個にまとめられる

〔リスト2-12〕
combine タイプ
COMMON の例
（その２）

```
 1: ;*****************************************************************
 2: ;
 3: ;　　機　　　能 :　　　combineタイプ COMMONの 例 そ の 2
 4: ;　　生　　　成 :　　　masm /ML common1;
 5: ;　　　　　　　　　　　masm /ML common2;
 6: ;　　　　　　　　　　　link /NOI/MAP common1 common2, common, common;
 7: ;
 8: ;*****************************************************************
 9:              PAGE     , 130
10:              PUBLIC   data3, data4, data5
11: dseg         SEGMENT COMMON 'DATA'                    ;COMMONの 指 定
12: data3        DW       3333h
13: data4        DW       4444h
14: dseg         ENDS
15:
16: dseg         SEGMENT COMMON 'DATA'                    ;COMMONの 指 定
17: data5        DW       5555h
18: dseg         ENDS
19:              END
```


〔リスト2-13〕combine タイプ COMMON による MAP ファイル

```
 LINK : warning L4021: no stack segment

  Start  Stop   Length Name                   Class
  00000H 0001EH 0001FH cseg                   CODE
  00020H 00025H 00006H dseg                   DATA

   Address         Publics by Name

  0002:0000        data3
  0002:0002        data4
  0002:0004        data5

   Address         Publics by Value

  0002:0000        data3
  0002:0002        data4
  0002:0004        data5

 Program entry point at 0000:0000
```


ます．

リスト2-12 において，セグメント dseg はまとめて一つのセグメントとして扱われ，data3～data5 には現われた順序でオフセット（メモリ割り当て）がつけられます．

これら二つのモジュールがリンクされると，セグメント dseg は**リスト2-12** において combine タイプ COMMON で定義されているためメモリ領域を共有し，同一のセグメント・ベースが与えられます．

リスト2-13（common.map）はリンク処理の結果得られた MAP ファイルです．同リストが示すように，data1 と data2 のメモリ領域には data3～data5 が割り当てられています．

リスト2-14 は，二つのモジュールをリンクして得られた実行モジュールの実行例を示しています．プログラムの動作を確認するために，ここでは SYMDEB を使っています．SYMDEB のサブコマンドについては，**表1-8**（22 ページ）に示しました．

① まず，デバッガ SYMDEB によってサンプル・プログラム common.exe を起動する．

② 次にレジスタ内容を確認する．

③ 命令コードを逆アセンブルして確認する．

④ BX レジスタには data1 の内容である 1111H が読み込まれるはずである．

⑤ CX レジスタには data2 の内容である 2222H が読み込まれるはずである．

⑥ 同様に DX レジスタ data3（3333H），SI レジスタに data4（4444H），DI レジスタに data5（5555H）が読み込まれるはずである．

⑦ オフセット 1BH まで実行して各レジスタを確認する．

⑧ BX レジスタには，data1（1111H）が読み込まれる

〔リスト2-14〕combine タイプ COMMON による common.exe の実行

```
R>symdeb common.exe ⏎ … デバッガによってcommon.exe を起動する ①
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-r ⏎ … レジスタ内容の確認 ②
AX=0000  BX=0000  CX=0147  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=6AC5  ES=6AC5  SS=6AD5  CS=6AD5  IP=0000   NV UP EI PL NZ NA PO NC
6AD5:0000 B8D76A        MOV     AX,6AD7
-u ⏎ … 逆アセンブルして確認 ③
6AD5:0003 8ED8          MOV     DS,AX
6AD5:0005 8B1E0000      MOV     BX,[0000] ←── data 1 (1111H) のハズ ④
6AD5:0009 8B0E0200      MOV     CX,[0002] ←── data 2 (2222H) のハズ ⑤
6AD5:000D 8B160000      MOV     DX,[0000] ←── data 3 (3333H) のハズ ┐
6AD5:0011 8B360200      MOV     SI,[0002] ←── data 4 (4444H) のハズ ├ ⑥
6AD5:0015 8B3E0400      MOV     DI,[0004] ←── data 5 (5555H) のハズ ┘
6AD5:0019 B44C          MOV     AH,4C                          ;'L'
6AD5:001B B000          MOV     AL,00
-g 1b ⏎ … 実行して各レジスタ内容の確認 ⑦
AX=4CD7  BX=3333  CX=4444  DX=3333  SP=0000  BP=0000  SI=4444  DI=5555
DS=6AD7  ES=6AC5  SS=6AD5  CS=6AD5  IP=001B   NV UP EI PL NZ NA PO NC
6AD5:001B B000          MOV     AL,00
-q ⏎
R>
```

data 1 (1111H) のハズが data 3 (3333H) になっている ⑧
data 2 (2222H) のハズが data 4 (4444H) になっている ⑨
data 3 は正しく読み込まれている ⑩
data 4 は正しく読み込まれている ⑪
data 5 は正しく読み込まれている ⑫

〔図2-9〕COMMON 指定によるセグメント配置

はずだったが，実際には data3(3333H) が読み込まれている．

⑨ CX レジスタには，data2(2222H) が読み込まれるはずだったが，実際には data4(4444H) が読み込まれている．

⑩ DX レジスタには，予定どおりに data3 が読み込まれている．

⑪ SI レジスタには，予定どおりに data4 が読み込まれている．

⑫ DI レジスタには，予定どおりに data5 が読み込まれている．

これらの結果から，セグメントの配置は図2-9 のように表すことができます．

▶ STACK

異なるモジュールで定義された複数のセグメントが，同じ name あるいは class で定義されている場合，こ

〔リスト2-15〕
combineタイプATの使用例

```
 1: ;*****************************************************************
 2: ;
 3: ;       機      能 :    combineタイプATの使用例
 4: ;       生      成 :    masm /ML at;
 5: ;                       link /NOI/MAP at,,at;
 6: ;
 7: ;*****************************************************************
 8:                PAGE     ,132
 9: aseg           SEGMENT  WORD PUBLIC 'CODE'
10:                ASSUME   CS:aseg
11:
12: start          PROC
13:                call     FAR PTR sub1              ;絶対アドレスのコール
14:                jmp      FAR PTR sub2              ;絶対アドレスへのジャンプ
15: start          ENDP
16: aseg           ENDS
17:
18: bseg           SEGMENT  AT 0F000h                 ;絶対アドレスを0F0000Hに設定
19:                ASSUME   CS:bseg
20:
21:                ORG      1000h                     ;オフセットの設定
22: sub1           PROC     FAR                       ;プロシージャの定義
23: sub1           ENDP
24:
25:                ORG      2000h                     ;オフセットの設定
26: sub2           PROC     FAR                       ;プロシージャの定義
27: sub2           ENDP
28: bseg           ENDS
29:                END
```


れらのセグメントはリンク時に連結されて一つのセグメントになる.

このcombine型は，スタック・セグメント用であることを除いて基本的にはPUBLICと変わらない．なお，このcombine型の指定を行うと，プログラム起動時のSP(スタック・ポインタ)には，これらのセグメントを合計した長さが初期設定される.

また,COMモデルの場合は，複数のセグメント指定ができないので，combine型がSTACKのセグメントを記述することは許されない(EXE2BINユーティリティでエラーが発生する).

逆にEXEモデルの場合には，どれかのモジュールでスタック・セグメントが指定されなければならない(この場合にはLINKからエラー・メッセージが表示される).

▶ AT <式>

式には，パラグラフ・アドレスを指定する．パラグラフ・アドレスとは，上位16ビットを指すアドレスである.

このcombine型は，アセンブル時にセグメントに絶対アドレスを割り付ける働きがある.なお,このATによる指定では，実際のオブジェクトがリンクされることはない.

このcombine型は,おもに絶対アドレスをコールしたり，そこにジャンプしたりする際のターゲット・ラベルのあるセグメントを仮定するのに使用される.

【ATのサンプル・プログラム】

リスト2-15(at.asm)は，combineタイプにATを使用した例を示しています.同リストのように,combineタイプにATを用いることによってセグメントasegからセグメントbsegのラベルを絶対アドレスで参照することが可能になります.

このときに，combineタイプATによって定義されたセグメントbseg中に命令コードやデータを記述しても，そのオブジェクトが実際にリンクされることはありません.

リスト2-16では，リンクによって得られた実行モジュールをロードしてそのオペランドの確認を行っています.ここでは,指定したとおりにセグメント0F000Hに対して絶対アドレスでアクセス可能になっています.

▶ MEMORY

COMMONとほとんど同じで,リンク時には同一のアドレスに配置され，さらにメモリの最上位にアドレスが配置される．このcombine型は，インテル社のMEMORY型との互換性を保つために用意されている.

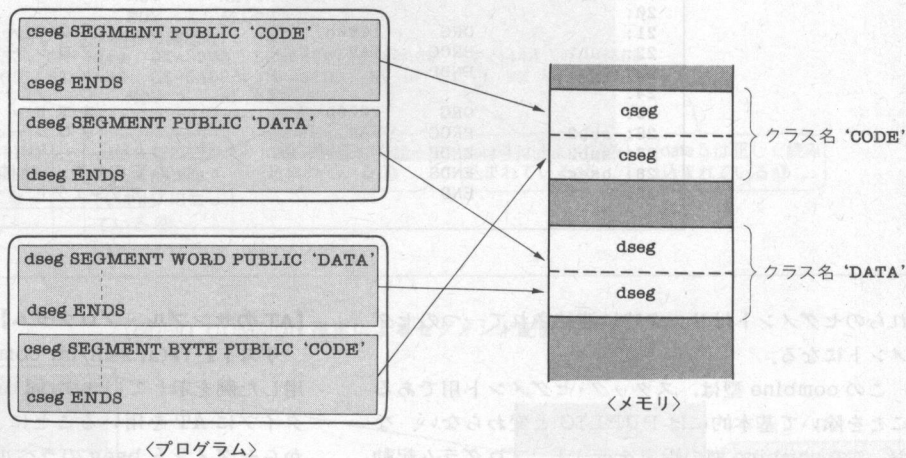
◆ useの指定

use型は80386プロセッサにおけるセグメント・ワード・サイズの指定を行います．セグメント・ワード・サイズとはセグメントのデフォルトのオペランドおよびアドレスのサイズのことをいいます.

use型には，USE16またはUSE32のいずれかを指定します．use型が関係するのは.386ディレクテ

〔リスト2-16〕得られた at.exe での命令コードの確認

```
R>symdeb at.exe ⏎ …デバッガによって at.exe を起動
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-u ⏎ …逆アセンブルして確認
6AD5:0000 9A001000F0     CALL     F000:1000
6AD5:0005 EA002000F0     JMP      F000:2000

-q ⏎

R>
```


〔図2-10〕
クラス名の指定による
セグメント配置

ィブを指定し，80386 用の命令とアドレッシング・モードを有効にしている場合に限定されます．

◆ classの指定

class(クラス名)は，'(シングル・クォーテーション)でくくられた文字列で表され，同じclassをもつセグメントは，LINKによって出会った順番にメモリ中に連続して展開されます．

すなわち，classや後述するGROUPディレクティブを使えば，それらの情報をLINKに渡すことによって，セグメントの配置をユーザが意識的に制御することが可能になります(図2-10)．

一つのソース・プログラム内で同じnameをもつ論理セグメントを，複数のSEGMENT～ENDSに分けて記述しても連続した一つのセグメントとみなされます．このときに，分割して書かれたSEGMENTステートメントに対して，異なるclassを指定することはできません．

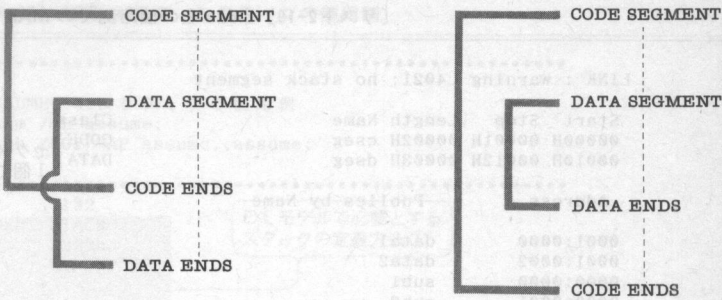
また，図2-11のようにSEGMENTステートメントはオーバラップすることはできませんが，異なるSEGMENTステートメントをネストさせることは可能です．このとき，ソース・プログラム上の論理セグメントと実際のメモリ上での物理セグメントとの配置は関係ありません．

【classのサンプル・プログラム】

リスト2-17(class1.asm)および**リスト2-18**(class2.asm)はクラス名の使用例を示しています．**リスト2-17**(class1.asm)において，セグメントcsegが最初に現われ，次にセグメントdsegが現われています．逆に**リスト2-18**(class2.asm)では，最初にセグメントdsegが現われ，その後にセグメントcsegが現われています．

これら二つのモジュールをリンクすることによって，**リスト2-19**(class.map)のように同一のセグメント名とクラス名をもつセグメントは一つのセグメントにまとめられます．また，そのセグメント配置の順序は，セグメントの現われた(定義された)順序になります．

そして，各セグメント内のラベル(コードやデータ)はリンクの順序にしたがって配置されることになります．

〔図2-11〕セグメントの配置

(a)　セグメントのオーバラップは不可

(b)　セグメントのネストは可

〔リスト2-17〕
class 名の使用例
（その1）

```
 1: ;*************************************************************
 2: ;
 3: ;    機    能 :    class名の使用例その1
 4: ;    生    成 :    masm /ML class1;
 5: ;                  masm /ML class2;
 6: ;                  link /NOI/MAP class1 class2, class, class;
 7: ;
 8: ;*************************************************************
 9:             PAGE      , 130
10:             PUBLIC    sub1, data1
11: cseg        SEGMENT   PUBLIC 'CODE'                ;PUBLICの指定
12:             ASSUME    CS:cseg
13:
14: sub1        LABEL     NEAR
15:             nop
16: cseg        ENDS
17:
18: dseg        SEGMENT   PUBLIC 'DATA'                ;PUBLICの指定
19: data1       DB        1
20: dseg        ENDS
21:             END
```

class 名

〔リスト2-18〕
class 名の使用例
（その2）

```
 1: ;*************************************************************
 2: ;
 3: ;    機    能 :    class名の使用例その2
 4: ;    生    成 :    masm /ML class1;
 5: ;                  masm /ML class2;
 6: ;                  link /NOI/MAP class1 class2, class, class;
 7: ;
 8: ;*************************************************************
 9:             PAGE      , 130
10:             PUBLIC    sub2, data2
11: dseg        SEGMENT   WORD PUBLIC 'DATA'           ;PUBLICの指定
12: data2       DB        2
13: dseg        ENDS
14:
15: cseg        SEGMENT   BYTE PUBLIC 'CODE'           ;PUBLICの指定
16:             ASSUME    CS:cseg
17:
18: sub2        LABEL     NEAR
19:             nop
20: cseg        ENDS
21:             END
```

〔リスト2-19〕class 名の使用例でのMAP ファイル

```
 LINK : warning L4021: no stack segment
  Start  Stop   Length Name                        Class
  00000H 00001H 00002H cseg                        CODE
  00010H 00012H 00003H dseg                        DATA

   Address         Publics by Name

  0001:0000        data1
  0001:0002        data2
  0000:0000        sub1
  0000:0001        sub2

   Address         Publics by Value

  0000:0000        sub1
  0000:0001        sub2
  0001:0000        data1
  0001:0002        data2
```

（CODE, DATA について）セグメント名と class 名が同一のセグメントは1個にまとめられる

（sub1, sub2 について）セグメント内のラベルはリンク時に現われた順番になる

● セグメント・レジスタの仮定

8086 CPU では，プログラムの実行時にデータをアクセスする場合には，その際に使用されるセグメント・レジスタに，その物理セグメントのセグメント・アドレスが設定されている必要があります．

このとき，使用されるセグメント・レジスタは，**表2-3** に示したようにアドレッシング・モードによってデフォルトのレジスタが決まっています．そして，このデフォルトのセグメント・レジスタの指すセグメント以外のセグメントをアクセスしたい場合には，“セグメント・オーバライド・プレフィックス”を命令コードの前に挿入します．これによって，デフォルト以外のセグメント・レジスタを使ってアクセスすることができます．

MASM では，ASSUME ディレクティブを用いることによって，ユーザがいちいちセグメント・オーバライドを記述することなしに，セグメント・オーバライド・プレフィックスを自動的に生成する機能をもっています．

ASSUME ディレクティブでは，プログラマによって論理セグメント/グループとセグメント・レジスタとの対応を指定します．これにより，MASM はメモリをアクセスする命令に出会うごとに，そのデータが定義されている論理セグメントがどのセグメント・レジスタを使用すべきかを決定し，セグメント・オーバライド・プレフィックスを必要に応じて自動的に生成します．

すなわち ASSUME ディレクティブは，CPU の物理的な各セグメント・レジスタが，ユーザの定義した論理的なセグメントのなかのどのレジスタに対応しているかを MASM に知らせる機能をもっていて，セグメントの仮定を行うディレクティブです．書式は次のようになります．

〔表2-3〕デフォルトで使用されるセグメント・レジスタ

アドレッシング・モード	オペランド書式	セグメント・レジスタ
ダイレクト	Disp16	DS
ベース	[BX]（+Disp8 または Disp16）	DS
	[BP]（+Disp8 または Disp16）	SS
インデックス	[SI]（+Disp8 または Disp16） [DI]（+Disp8 または Disp16）	DS
ベース・インデックス	[BX] [SI]（+Disp8 または Disp16） [BX] [DI]（+Disp8 または Disp16）	DS
	[BP] [SI]（+Disp8 または Disp16） [BP] [DI]（+Disp8 または Disp16）	SS

ASSUME の構文

ASSUME segmentregister:name [,…]
　または
ASSUME segmentregister:NOTHING
　または
ASSUME NOTHING

segmentregister には，CPU の物理的なセグメント・レジスタである CS，DS，ES，SS の四つのうちのいずれかのセグメント・レジスタを指定します．

name には，SEGMENT ディレクティブで指定した segmentname あるいはグループ名を指定します．ここで注意すべきことは，ASSUME ディレクティブは単なるセグメントの仮定を行うだけであって，実際にプログラムがロードされたときの，CPU の物理的なセグメント・レジスタ内容には何ら関与しているものではないということです．

したがって，これらのレジスタで設定されるべき物理的な値は，ユーザ・プログラムの中でたとえば mov 命令などを用いて，必要な値が設定されるように事前

〔リスト2-20〕ASSUMEディレクティブの使用例

```
                ;*************************************************************
                ;
                ;       機    能:       ASSUMEディレクティブの使用例
                ;       生    成:       masm /ML assume;
                ;                       link /NOI/MAP assume,,assume;
                ;
                ;*************************************************************
                                PAGE    ,132
0000                    sseg    SEGMENT STACK
0000    2222            data2   DW      2222h
0002    0040[                   DW      64 DUP (?)
        ????
                ]
0082                    sseg    ENDS

0000                    cseg    SEGMENT
                                ASSUME  CS:cseg, DS:dseg, SS:sseg

0000                    start:
0000    B8 ---- R               mov     ax, dseg
0003    8E D8                   mov     ds, ax
0005    A1 0000 R               mov     ax, data1
0008    36: 8B 1E 0000 R        mov     bx, data2
000D    B4 4C                   mov     ah, 4Ch
000F    B0 00                   mov     al, 00h         ;リターン・コード
0011    CD 21                   int     21h             ;プログラム終了
0013                    cseg    ENDS

0000                    dseg    SEGMENT
0000    1111            data1   DW      1111h
0002                    dseg    ENDS
                                END     start
```


に記述されていなければなりません．

【ASSUMEのサンプル・プログラム】

リスト2-20(assume.lst)は，ASSUMEディレクティブの使用例であり，MASMから出力されたアセンブル・リストです．

① EXEモデルでは，必ずスタック・セグメントを必要とする．

② DSレジスタは，セグメントdsegを指しているものと仮定する．

③ 同様に，SSレジスタはセグメントssegを指しているものと仮定する．

④ ASSUMEディレクティブは，セグメントの仮定を行っているだけなので，プログラム内では実際にセグメント・レジスタの設定を行う必要がある．ただし，EXEモデルのプログラムでは，SSレジスタはスタック・セグメントに自動的に初期設定される．

⑤ セグメントdsegに属するdata1のアクセス．ここでは，デフォルトのセグメント・レジスタとしてDSレジスタが使用される．

⑥ セグメントssegに属するdata2のアクセス．ここでは，ASSUMEディレクティブによってSSレジスタがセグメントssegを指すように仮定されていて，実際にプログラムがロードされた時点で，仮定されたとおりにSSレジスタがセグメントssegに初期設定されている．したがって，デフォルト・セグメントのDSレジスタではアクセスできない．

⑦ しかし，MASMはASSUMEディレクティブによってdata2の属するセグメントssegは，SSレジスタに対応していることが認識できるので，セグメント・オーバライド・プレフィックスを自動的に付加している．

● セグメントのグループ化

グループとは，いくつかの論理セグメントをまとめたものです．GROUPディレクティブを用いることによって，プログラムは複数の論理セグメントに対してセグメント・レジスタの内容を変えないでアクセスすることが可能になります．

このGROUPディレクティブを使用してまとめられたセグメントでは，セグメント間の分岐はNEAR(16ビット長)として扱われるので，オブジェクトの生成効率がよくなります．GROUPディレクティブの書式は次のようなものです．

GROUPの構文

```
name GROUP segment [,…]
```

nameには，SEGMENTで指定したsegment-name，またはSEG変数やSEGラベルを指定することができます(演算子参照)．SEG変数/ラベルの場合は，その変数またはラベルが定義されているセグメント名が指定されます．

ここで，GROUPディレクティブにおいて指定されたsegmentの順序は，それらの論理セグメントのメモリ配置にはなんら影響を与えません．また，GROUPディレクティブは，その nameが参照される以前に定義されていなければなりません．

【GROUPのサンプル・プログラム】

リスト2-21(group1.asm)，および**リスト2-22**(group2.asm)は，別々にアセンブルされたモジュール内のセグメントを，GROUPディレクティブによって一つのセグメントにまとめて使用するプログラム例を示しています．

リスト2-21(group1.asm)において，セグメントcseg2およびセグメントdseg2は，他のモジュール(**リスト2-22**)内にある論理セグメントとのグループを定義するために，このファイルの最初でダミーのセグメント定義を行っています．

セグメントcseg1とcseg2は，GROUPディレクティブによって一つのセグメントにまとめられ，グループ名cgroupとして扱うことができます．同様に，セグメントdseg1とdseg2も一つのセグメントにまとめられ，グループ名dgroupとして扱うことができます．

リスト2-21の26行目において，AXレジスタにはdgroupの配置されるセグメント・アドレス値がロードされ，27行目でDSレジスタに設定されます．これによって，dgroup内のデータはデフォルトのセグメント・レジスタからのオフセットとしてアクセス可能となります．

リスト2-23(group.map)は，アセンブル/リンクによって得られたMAPファイルの内容です．同リストにおいて，セグメントcseg2とcseg1はcgroupにグループ化され，セグメントdseg2とdseg1はdgroupにグループ化されているのが確認できます．

また，セグメントcseg2は**リスト2-21**においてセグメントcseg1の前にダミー定義されているため，実際のオブジェクト・コードもセグメントcseg2内で記述されたものが下位アドレスに配置されます(**図2-12**)．

リスト2-24は，得られた実行モジュールgroup.exe

〔リスト2-21〕 GROUPディレクティブの使用例(その1)

```
 1: ;*************************************************************
 2: ;
 3: ;  機     能:   group名の使用例その1
 4: ;  生     成:   masm /ML group1;
 5: ;               masm /ML group2;
 6: ;               link /NOI/MAP group1 group2, group, group;
 7: ;
 8: ;*************************************************************
 9:             PAGE     , 130
10: cseg2       SEGMENT PUBLIC 'CODE'           ;セグメントのダミー定義
11: cseg2       ENDS
12: dseg2       SEGMENT PUBLIC 'DATA'           ;セグメントのダミー定義
13: dseg2       ENDS
14: stack       SEGMENT STACK                   ;スタック・セグメントの定義
15:             DW      128 DUP (?)
16: stack       ENDS
17:
18: cgroup      GROUP   cseg1, cseg2   ←── 1個のセグメントcgroupとして扱うことができる
19: dgroup      GROUP   dseg1, dseg2   ←── 1個のセグメントdgroupとして扱うことができる
20:             PUBLIC  start, data1
21:             EXTRN   msg_out:NEAR, data2:WORD
22: cseg1       SEGMENT PUBLIC 'CODE'           ;PUBLICの指定
23:             ASSUME  CS:cgroup, DS:dgroup
24:
25: start       PROC
26:             mov     ax, SEG dgroup
27:             mov     ds, ax                  ;DSレジスタの設定
28:             call    msg_out                 ;メッセージの表示
29:             mov     bx, data1
30:             mov     cx, data2
31:             mov     ah, 4Ch
32:             mov     al, 00h                 ;リターン・コード
33:             int     21h                     ;プログラム終了
34: start       ENDP
35: cseg1       ENDS
36:
37: dseg1       SEGMENT PUBLIC 'DATA'           ;PUBLICの指定
38: data1       DW      1111h
39: dseg1       ENDS
40:             END     start
```

の実行例です．同リストの実行例が示すように各データへのアクセスは正常に行われていることが確認できます．

● セグメントの簡略化定義

MASM ver.5.1 では，セグメント定義のために簡略化システムを新たに導入しました．このセグメントの簡略化定義のための新しいディレクティブとして，メモリ・モデルを宣言するための .MODEL，およびセグメントの定義を行う表2-4 に示したディレクティブが用意されています．

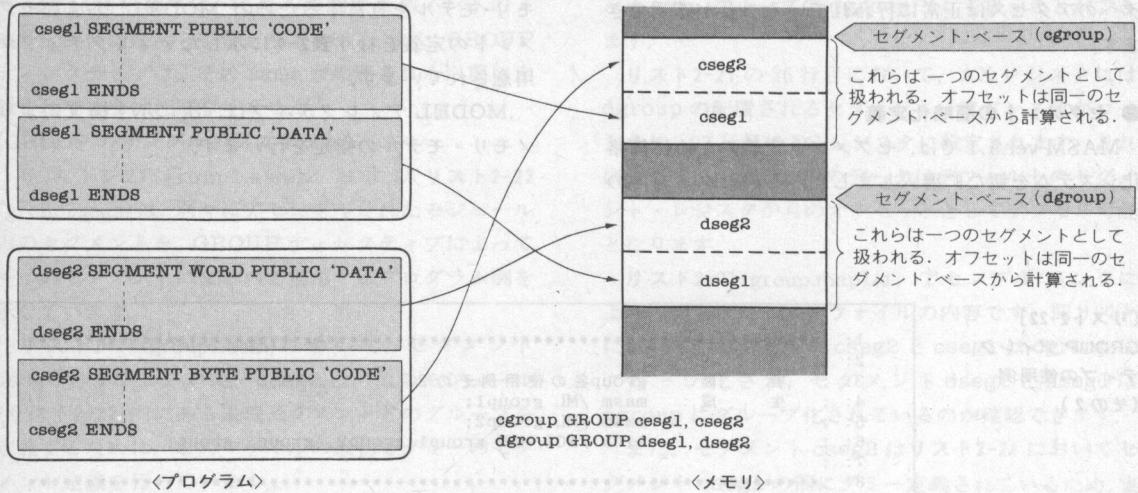
.MODEL ディレクティブは，次に示す構文によりメモリ・モデルの指定を行います．

〔リスト2-22〕 GROUP ディレクティブの使用例（その2）

```
 1: ;*****************************************************************
 2: ;
 3: ;  機      能:    group名の使用例その2
 4: ;  生      成:    masm /ML group1;
 5: ;                 masm /ML group2;
 6: ;                 link /NOI/MAP group1 group2, group, group;
 7: ;
 8: ;*****************************************************************
 9:               PAGE      , 130
10:               PUBLIC    msg_out, data2
11: dseg2         SEGMENT PUBLIC 'DATA'          ;PUBLICの指定
12: data2         DW        2222h
13: msg           DB        'GROUPディレクティブのテスト・プログラム'
14:               DB        0Dh, 0Ah, '$'
15: dseg2         ENDS
16:               これらのセグメントは他のモジュールでグループ化されている
17: cseg2         SEGMENT PUBLIC 'CODE'          ;PUBLICの指定
18:               ASSUME    CS:cseg2
19:
20: msg_out       PROC
21:               lea       dx, msg
22:               mov       ah, 09h
23:               int       21h                  ;文字列表示
24:               ret
25: msg_out       ENDP
26: cseg2         ENDS
27:               END
```

〔リスト2-23〕 LINK から出力された MAP ファイル

```
 Start  Stop   Length Name                   Class
 00000H 00008H 00009H cseg2                  CODE  }グループ cgroup
 00010H 00025H 00016H cseg1                  CODE  }
 00030H 0005BH 0002CH dseg2                  DATA  }グループ dgroup
 00060H 00061H 00002H dseg1                  DATA  }
 00070H 0016FH 00100H stack

 Origin   Group
 0000:0   cgroup
 0003:0   dgroup

  Address         Publics by Name

 0003:0030       data1
 0003:0000       data2
 0000:0000       msg_out
 0000:0010       start

  Address         Publics by Value

 0000:0000       msg_out
 0000:0010       start
 0003:0000       data2
 0003:0030       data1

Program entry point at 0000:0010
```

〔図2-12〕グループ化によるセグメントの配置

〔リスト2-24〕 GROUP ディレクティブを用いたプログラム group.exe の実行

```
R>symdeb group.exe ⏎ … デバッガを用いてプログラム group.exe を起動
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-u ⏎ … 逆アセンブルして確認する
6AD5:0010 B8D86A        MOV    AX,6AD8
6AD5:0013 8ED8          MOV    DS,AX
6AD5:0015 E8E8FF        CALL   0000
6AD5:0018 8B1E3000      MOV    BX,[0030] ←— data 1 (セグメント dseg 1) へのアクセス
6AD5:001C 8B0E0000      MOV    CX,[0000] ←— data 2 (セグメント dseg 2) へのアクセス
                                                                ;'L'
6AD5:0020 B44C          MOV    AH,4C
6AD5:0022 B000          MOV    AL,00
6AD5:0024 CD21          INT    21
-g 24 ⏎ … 実行して確認
GROUPディレクティブのテスト・プログラム
AX=4C00  BX=1111  CX=2222  DX=0002  SP=0100  BP=0000  SI=0000  DI=0000
DS=6AD8  ES=6AC5  SS=6ADC  CS=6AD5  IP=0024   NV UP EI PL NZ NA PO NC
6AD5:0024 CD21          INT    21  :Terminate a Process
-q ⏎
        data 1 の   data 2 の
R>      内容        内容
```

〔表2-4〕簡略化セグメント・ディレクティブ

セグメント・ディレクティブ	セグメントの機能
.STACK [size]	スタック・セグメント
.CODE [name]	コード・セグメント
.DATA	初期化済み near データ・セグメント
.DATA?	未初期化 near データ・セグメント
.FARDATA [name]	初期化済み far データ・セグメント
.FARDATA? [name]	未初期化 far データ・セグメント
.CONST	定数データ・セグメント

.MODEL の構文

.MODEL memorymodel [,language]

memorymodel には，**表2-5** に示したモデルを指定することができます．高級言語で作成したプログラムから呼ばれるアセンブリ言語のサブルーチンを作成する場合，この memorymodel は，該当するコンパイラが使用するメモリ・モデルと一致させる必要があります．単独で実行するアセンブリ・プログラムの場合は，どのメモリ・モデルでも使用可能です．

language は，プロシージャの名前や呼び出し/リターンなどの手順を，指定された言語に合わせるように MASM に指示するために用いられます．たとえば，language に C を指定するとプロシージャ名の先頭に

は“_(アンダ・スコア)”が自動的に付加されます．また, languageが指定されると, すべてのプロシージャ名はPUBLIC宣言されて外部モジュールから参照可能となります．

languageは, プロシージャの呼び出しで引数を渡す際のスタック・フレームの生成にも影響を与えます．たとえば, languageにFORTRANやPASCALを指定すると, 引数は現れた順にスタックに積まれているものと仮定します(最後の引数がスタックのトップにある)．languageにCを指定すると, 引数は逆の順序でスタックに積まれているものと仮定します．

プログラムでセグメント名を参照するには, **表2-4**のディレクティブ名の先頭に“@(アット・マーク)”を付加して参照することができます．

【簡略化セグメント定義のサンプル・プログラム】

リスト2-25(seg.asm)は, 簡略化されたセグメントの定義例を示しています．同リストにおいて, .MODELディレクティブでは, メモリ・モデルにSMALLを指定し, 使用言語にはCを指定しています．この指定によってスタック・フレームがC言語の仕様に合わせられます．

次に, .STACKディレクティブではスタック領域と

〔表2-5〕メモリ・モデル

メモリ・モデル	機　　能
SMALL	データおよびコードは，それぞれ一つのセグメントにまとめられ，64 Kバイト以下でなければならない．すべてのコードとデータはデフォルトでnearとして扱われる．高級言語ではCだけがこのモデルをサポートしている．
MEDIUM	データは一つのセグメントにまとめられ，64 Kバイト以下であるが，コードは64 Kバイト以上になる場合がある．したがって，デフォルトでデータはnearで扱われ，コードはfarで扱われる．
COMPACT	コードは一つのセグメントにまとめられ，64 Kバイト以下であるが，データは64 Kバイト以上になる場合がある(ただし配列は64 Kバイトを越えられない)．したがって，コードはnearで扱われるがデータはfarで扱われる．
LARGE	コードおよびデータともに64 Kバイト以上になる場合がある(ただし配列は64 Kバイト以下)．したがって，コードとデータはデフォルトでfarとして扱われる．
HUGE	コードおよびデータともに64 Kバイト以上になる場合がある．また，配列は64 Kバイトを越えてもかまわない．したがって，コードとデータはfarとして扱われ，配列要素へのポインタもfarとして扱わなければならない．

〔リスト2-25〕簡略化セグメントの定義例

```
 1: ;*******************************************************************
 2: ;
 3: ;    機    能 :  簡略化セグメントの定義例
 4: ;    生    成 :  masm /ML seg;
 5: ;                link /NOI/MAP seg,,seg;
 6: ;
 7: ;*******************************************************************
 8:           PAGE      ,132
 9:           .MODEL    SMALL, C              ;メモリ・モデル／言語定義
10:           .STACK    256                   ;スタック・セグメント
11:           .DATA                           ;データ・セグメント
12: data1     DW        1111h
13: data2     DD        22223333h
14:           .CODE                           ;コード・セグメント
15:
16: start     PROC
17:           mov       ax, @data
18:           mov       ds, ax                ;DSレジスタ初期化
19:           lea       bx, data2
20:           push      bx                    ;引数 2
21:           mov       ax, data1
22:           push      ax                    ;引数 1
23:           call      sub1
24:           add       sp, 4
25:           mov       ah, 4Ch
26:           mov       al, 00h               ;リターン・コード
27:           int       21h                   ;プログラム終了
28: start     ENDP
29:
30: sub1      PROC      arg1, arg2:PTR
31:           mov       bx, arg2
32:           les       di, [bx]              ;data2
33:           mov       ax, arg1              ;data1
34:           ret
35: sub1      ENDP
36:           END       start
```

して256バイトの確保を行っています．12行目や13行目のdata1およびdata2などのラベル名には自動的に“_”が付加されます．また，16行目のラベルstartなども同様ですが，プロシージャ名の場合は自動的にPUBLIC宣言され，他のモジュールから外部参照が可能になります．

17行目では，AXレジスタにデータ・セグメントのセグメント値をロードしてDSレジスタを初期化しています．

19行目から22行目では，二つの引数をスタックにプッシュしてサブルーチンsub1の引数としています．サブルーチンsub1では，これらのスタックで渡された引数を取り出しています．引数arg2はdata2へのポインタであり，data2はダブル・ワードのデータなので，その内容をES：DIレジスタにロードしています．また，引数arg1は，ワード・データなのでAXレジスタにロードしています．

● プロシージャの定義

MASMでは，PROCディレクティブで始まりENDPディレクティブで終わるサブルーチンを“プロシージャ”と呼んでプログラムの構造化を促進しています．

一方，8086 CPUでは，サブルーチン・コールを行うCALL命令と，そのサブルーチンから戻るRET命令にそれぞれNEAR(64Kバイト以内)とFAR(64Kバイト以上：セグメント・レジスタ変更)の2通りの方法が用意されています．

MASMでは，プロシージャの宣言(記述)を行う際にNEARとFARの型をもたせることによって，このNEARとFARのうちのどの命令コードを生成すべきかを決定しています．

PROCの構文①

```
label PROC [NEAR|FAR]
    ⋮
label ENDP
```

labelはプロシージャの名前として定義され，そのプロシージャの先頭を表すラベルとして使用されます．プロシージャの型属性には，NEARまたはFARを指定することができ，デフォルトではNEARが指定されます．この型属性は，プロシージャとして呼ばれた際にリターンするRET命令のコード生成に関係します．

【PROCのサンプル・プログラム①】

リスト2-26(proc.lst)は，PROCディレクティブの使用例を示しています．同リストにおいて，呼び出し

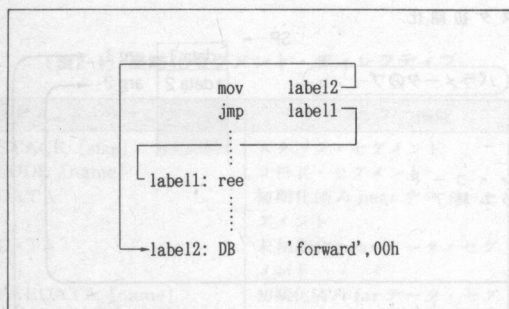
● 前方参照 ●

ソフトウェアの解説書やマニュアルには，「前方参照」という言葉がよく出てきます．前方参照とは，**リストA**のようにオペランドに参照すべきラベルが出てきても，そのラベルがまだラベルとして認識されていない場合をいいます．

日本語的な感覚で考えると，前方参照というのは同リストとは逆に，ラベルが出てくる以前にラベルとして認識される場合，すなわちリストの前のほうに出てくることのような気がします．

この点では筆者も疑問に思っていましたが，文献によると，語源はforword(進行方向)からきているのだそうです．進行方向であれば，すなわちアドレスの増える方向ですから，まさに「前方参照」といっても納得できます．

すなわち，「前方」とはプログラマの考えている前方ではなく，あくまでもコンピュータが実行していくときの前方のことなのです．

〔リストA〕前方参照の例

側のオペランドがFARプロシージャとNEARプロシージャでは，呼び出しの機械語コードに相違が出てきます．また呼び出されるプロシージャでも，FARとNEARではリターン命令の機械語コードが異なります．

FARプロシージャを前方参照する場合は，演算子(後述：77ページ)を利用してそのディスタンス(長さ)をMASMに知らせてやらなければなりません．

● PROCディレクティブによるパラメータ宣言

MASM ver.5.1のPROCディレクティブでは，上記のプロシージャ名の宣言に加えて，高級言語とのインターフェースを簡単に実現できる機能が拡張されています．書式は次のとおりです．

PROCの構文②

```
label PROC [NEAR|FAR] [USES [reglist] ,] [arguments]
    :
label ENDP
```

labelはプロシージャの名前であり，もし.MODELディレクティブで言語としてCが指定された場合は，labelの先頭に“_”が自動的に付加されます．

reglistは，プロシージャが使用するために退避すべきレジスタのリストです．リスト中のレジスタは空白またはタブで区切ります．この機能によって，そのプロシージャ内におけるCPUニーモニックのPUSH命令やPOP命令を省略することが可能となっています．

argumentsは，プロシージャにスタックで渡される引数です．

argumentsの構文は次のようになっています．

argumentname[: [NEAR|FAR]PTR] type…

ここで，argumentnameは引数の名前です．typeは引数の型を表し，WORD/DWORD/FWORD/QWORD/TBYTEまたはSTRUCディレクティブで宣言されたストラクチャの名前を指定します．typeのデフォルトはWORDになります．

〔リスト2-26〕PROCディレクティブの使用例

```
                ;**************************************************************
                ;
                ;       機  能 :   PROCディレクティブの使用例
                ;       生  成 :   masm /ML proc;
                ;                  link /NOI/MAP proc,,proc;
                ;
                ;**************************************************************
                                PAGE      ,132
 0000                   cseg1   SEGMENT                 ;セグメント(cseg1)定義
                                ASSUME    CS:cseg1

 0000                   sub1    PROC      FAR           ;FARプロシージャ
 0000  CB                       ret
 0001                   sub1    ENDP
 0001                   cseg1   ENDS

 0000                   cseg2   SEGMENT                 ;セグメント(cseg2)定義
                                ASSUME    CS:cseg2

 0000                   start   PROC                    ;NEARプロシージャ
 0000  9A 0000 ---- R           call      sub1          ;FARコール
 0005  E8 0013 R                call      sub2          ;NEARコール
 0008  9A 0000 ---- R           call      FAR PTR sub3  ;FARコール(前方参照)
 000D  B4 4C                    mov       ah, 4Ch
 000F  B0 00                    mov       al, 00h       ;リターン・コード
 0011  CD 21                    int       21h           ;プログラム終了
 0013                   start   ENDP

 0013                   sub2    PROC      NEAR          ;NEARプロシージャ
 0013  C3                       ret
 0014                   sub2    ENDP
 0014                   cseg2   ENDS

 0000                   cseg3   SEGMENT                 ;セグメント(cseg3)定義
                                ASSUME    CS:cseg3

 0000                   sub3    PROC      FAR           ;FARプロシージャ
 0000  CB                       ret
 0001                   sub3    ENDP
 0001                   cseg3   ENDS
                                END       start
```

【PROCのサンプル・プログラム②】

リスト2-27(arg.lst)は，PROCディレクティブによるパラメータ宣言の使用例を示しています．同リストでは，.MODELディレクティブによってメモリ・モデルをSMALLに，言語インターフェースをCに設定しています．このため，スタックにプッシュする引数の順序は，引数3，2，1の順に行います．

ここで，引数1(arg1)はワード・データであり，引数2(arg2)はワード・データへのポインタであり，引数3(arg3)はFARプロシージャへのポインタです．

プロシージャsub1では，これらの引数をそれぞれワード・データ，NEARポインタ，FARポインタとして受け取り，data1，data2へのアクセス，およびarg3(sub1)へのFARコールの実現方法を示しています．

なお，同リストではプロシージャ内で使用するレジスタのPUSH命令やPOP命令(PROCディレクティブのUSESフィールドで指定)がリスティングされていませんが，MASMの/LAオプションを用いることによってアセンブル・リストに出力させることも可能です．

〔リスト2-27〕PROCディレクティブによるパラメータ宣言

```
                          ;*************************************************************
                          ;
                          ;    機    能:    PROCディレクティブによるパラメータの宣言
                          ;    生    成:    masm /ML arg;
                          ;                 link /NOI/MAP arg,,arg;
                          ;
                          ;*************************************************************
                                          PAGE   ,132
                                          .MODEL SMALL, C
                                          EXTRN  sub3:FAR ←─外部参照によるFARプロシージャの宣言
                                          .STACK ←─スタック・セグメントの確保
                                          .DATA ←─データ・セグメント
0000  1111                  data1         DW     1111h
0002  2222                  data2         DW     2222h
                                          .CODE ←─コード・セグメント
0000                        start         PROC
0000  B8 ---- R                           mov    ax, @data ←─データ・セグメント値
0003  8E D8                               mov    ds, ax          ;DSレジスタの初期化
0005  B9 ---- E                           mov    cx, SEG sub3    ;sub3のセグメント
0008  8D 16 0000 E                        lea    dx, sub3        ;sub3のオフセット
000C  51                                  push   cx              FARポインタのロード
000D  52                                  push   dx              ;引数3のプッシュ
000E  8D 1E 0002 R                        lea    bx, data2       ;data2へのポインタ
0012  53                                  push   bx              ;引数2のプッシュ
0013  A1 0000 R                           mov    ax, data1       ;data1のロード
0016  50                                  push   ax              ;引数1のプッシュ
0017  E8 002C R                           call   sub1            ;NEARプロシージャのコール
001A  83 C4 08                            add    sp, 8           ;スタック・ポインタの整合
001D  50                                  push   ax              ;引数1のプッシュ
001E  9A 0044 ---- R                      call   FAR PTR sub2
0023  83 C4 02                            add    sp, 2           ;スタック・ポインタの整合
0026  B4 4C        (FARコール)            mov    ah, 4Ch
0028  B0 00                               mov    al, 00h         ;リターン・コード
002A  CD 21                               int    21h             ;プログラム終了
002C                        start         ENDP
      (オブジェクトの中にPUSH命令が埋め込まれる)  (レジスタ・リスト)  (引数の型宣言)
002C                        sub1          PROC   USES ax bx es di,arg1:WORD,arg2:PTR,arg3:FAR PTR
0033  8B 46 04                            mov    ax, arg1        ;引数1(1111h)
0036  8B 5E 06                            mov    bx, arg2        ;引数2(data2へのNEARポインタ)
0039  8B 0F                               mov    cx, [bx]        ;data2のアクセス
003B  FF 5E 08                            call   arg3            ;sub3(FARプロシージャ)の実行
0043  C3                                  ret
0044                        sub1          ENDP                   (FARコール)

0044                        sub2          PROC   FAR USES ax, arg1:WORD
0048  8B 46 06                            mov    ax, arg1        ;引数1
004D  CB                                  ret
004E                        sub2          ENDP   (FARプロシージャ)
                                          END    start
      (オブジェクトの中にPOP命令が埋め込まれる)
```

ラベル

ラベルは，プログラムの分岐先，すなわち命令コードを含むロケーション(アドレス)を表し，JMP命令やCALL命令などによって参照されます．ラベルは以下の方法で定義することができます．

(1) LABELディレクティブによる定義
(2) “:”による定義
(3) EQU，“=”ディレクティブによる定義
(4) PROCディレクティブによるプロシージャ名としての定義
(5) ラベルの属性をもつEXTRNディレクティブによる定義

以下に，それぞれについて説明します．

● LABELディレクティブによる定義

LABELディレクティブを用いるとシンボルの型を自由に指定することができます．

name LABEL distance

distanceには，NEAR/FAR/BYTE/WORD/DWORD/QWORDのいずれかを指定します．nameがラベル名の場合は，distanceはNEARかFARになります．nameが変数名の場合は，distanceはBYTE/WORD/DWORD/QWORD/TBYTE，ストラクチャ名またはレコード名などで指定します．

ここで，distanceにNEARまたはFARを指定する場合は，ASSUMEディレクティブによってCSレジスタにその論理セグメント名かグループ名が指定されていなければなりません．

また，LABELディレクティブの変数への適用は，メモリ領域を確保することなしに変数を定義する場合や，変数のアクセスを定義された型とは異なる型で行うために使用されます．

● NEARラベルの定義

NEARラベルの場合は，“:”を用いて簡略化して定義することができます．

name:

“:”は，LABEL NEARの省略形であり，distanceにNEARをもつラベルの生成を行います．ここで，LABEL NEARの場合は，その行に他の命令を書くことができませんが，name:は，命令の前や名前フィールドをもたないディレクティブの名前フィールドに記述することが許されます．

また，簡略化セグメントの.MODELディレクティブを用いて言語の指定を行った場合，name:は，そのプロシージャ内のみで有効であり，他のプロシージャで同一のname:を使用してもさしつかえありません．

【ラベルのサンプル・プログラム】

リスト2-28(label.lst)は，ラベルの定義例を示しています．同リストにおいて，ラベルdata1とsub1はEXTRNディレクティブによってそれぞれWORDおよびFARラベルとして宣言されます．また，data2はディレクティブDDによってダブル・ワードとなり，data3はLABELディレクティブによってバイトの属性をもって定義されています．

また，sub2およびsub4はPROCディレクティブによって，それぞれNEARおよびFARラベルとして宣言されています．ラベルsub3は，LABELディレクティブによってNEARレベルとして宣言されています．

ここで，NEARラベルloop1はプロシージャsub2とsub4で同一のラベル名で定義されていますが，ディレクティブ.MODELによって言語のタイプをCとして指定しているためエラーとならず，そのプロシージャ内のみで有効な局所的ラベルとして扱われます．

外部参照

MS-DOSでは，別々にアセンブルされたオブジェクト・ファイルの単位をモジュールと呼んでいますが，あるモジュールから他のモジュールの中にある変数やラベル，あるいはプロシージャなどを参照することを外部参照といいます．

この外部参照をサポートするディレクティブにPUBLICとEXTRNディレクティブがあり，この二つのディレクティブは対で使用されます．

PUBLICディレクティブは，LINKに対してその名前が他のモジュールから参照可能な名前であることを宣言します．

EXTRNディレクティブは，逆に他のモジュールで定義された名前の参照を指定するもので，その名前の型を宣言します．

PUBLICの構文

PUBLIC name [,…]

PUBLICディレクティブは，ソース・ファイル内で定義されたnameを他のモジュールから参照可能とするための宣言を行います．nameは，そのソース・ファイル内で定義されている数値や変数，あるいはラベルに対応する名前でなければなりません．

ここでnameは，レジスタ名やEQUディレクティブで定義された浮動小数点数値，および2バイトを越える整数を表す名前は許されないので注意が必要です．

EXTRNの構文

EXTRN name:type [,…]

nameには型属性であるtypeを指定します．name

〔リスト2-28〕ラベル定義の例

```
                    ;*************************************************************
                    ;
                    ;       機    能 :      ラベルの定義例
                    ;       生    成 :      masm /ML label;
                    ;                       link /NOI/MAP label ??,,label;
                    ;
                    ;*************************************************************
                                    PAGE    , 132
                                    EXTRN   data1:WORD, sub1:FAR
                                    .MODEL  SMALL, C            ;メモリ・モデル／言語の定義
                                    .DATA                       ;データ・セグメント
0000  22221111          data2       DD      22221111h
0004                    data3       LABEL   BYTE
0004  3333                          DW      3333h

                                    .CODE                       ;コード・セグメント
0000                    start       PROC
0000  B8 ---- R                     mov     ax, @data
0003  8E D8                         mov     ds, ax              ;DSレジスタの初期化
0005  9A 0000 ---- E                call    sub1                ;外部参照（FARラベル）
000A  E8 0026 R                     call    sub2                ;NEARラベル
000D  E8 002A R                     call    sub3                ;NEARラベル
0010  9A 002D ---- R                call    FAR PTR sub4        ;FARラベル
0015  A1 0000 E                     mov     ax, data1           ;ワード属性
0018  C4 1E 0000 R                  les     bx, data2           ;ダブル・ワード属性
001C  8A 0E 0004 R                  mov     cl, data3           ;バイト属性
0020  B4 4C                         mov     ah, 4Ch
0022  B0 00                         mov     al, 00h             ;リターン・コード
0024  CD 21                         int     21h                 ;プログラム終了
0026                    start       ENDP

0026                    sub2        PROC    NEAR                ;NEARラベル
0026  33 C9                         xor     cx, cx
0028                    loop1:
0028  E2 FE                         loop    loop1
002A                    sub3        LABEL   NEAR                ;NEARラベル
002A  33 C0                         xor     ax, ax
002C  C3                            ret
002D                    sub2        ENDP

002D                    sub4        PROC    FAR                 ;FARラベル
002D                    loop1:
002D  E2 FE                         loop    loop1
002F  CB                            ret
0030                    sub4        ENDP
                                    END     start
```


が変数名の場合には，typeに対してデータの長さを指示するBYTE/WORD/DWORD/FWORD/QWORD/TBYTEのいずれかを指定します．nameがラベル名の場合には，そのラベルのディスタンスを決めるNEAR/FARのいずれかを指定します．

また，nameがEQUディレクティブで定義された名前の場合には，その数値や文字を表すABSなどを指定することができます．

ラベルを宣言するEXTRNディレクティブは，ソース・プログラム内のどこに書いてもかまいませんが，変数を宣言するEXTRNディレクティブが，ある論理セグメントの定義ブロック(SEGMENT~ENDS)の中に書かれた場合，そこで宣言されたnameは，その論理セグメントまたはグループ内に存在するものとみなされます．

【外部参照のサンプル・プログラム】

リスト2-29(extrn1.asm)および**リスト2-30**(extrn2.asm)は，外部参照を行う場合のプログラミング例を示しています．

リスト2-29(extrn1.asm)において，EXTRNディレクティブによって宣言されたラベルは，それぞれの属性をもって他のモジュールで定義されていることを表しています．EXTRN宣言を行うと，これら他のモジュールで定義されているラベルも，そのモジュール内で定義されているラベルとまったく同様に扱うことが可能となります．

リスト2-30(extrn2.asm)では，それら外部参照を可能にすべきラベルをPUBLICディレクティブを用いて宣言しています．PUBLICディレクティブでは，ディスタンスを指定する必要がなく，すべてのラベルを宣言することができます．

〔リスト2-29〕外部参照のディレクティブの使用例(その1)

```
 1: ;*************************************************************
 2: ;
 3: ;   機      能 :   外部参照のプログラミング例その1
 4: ;   生      成 :   masm /ML extrn1;
 5: ;                  masm /ML extrn2;
 6: ;                  link /NOI/MAP extrn1 extrn2, extrn, extrn;
 7: ;
 8: ;*************************************************************
 9:             PAGE    ,132
10:             .MODEL  SMALL, C                    ;メモリ・モデル/言語定義
11:
12:             EXTRN   sub1:FAR, sub2:NEAR
13:             EXTRN   data1:WORD, data2:DWORD ◄──(他のモジュールに定義されているラベル)
14:             EXTRN   _data3:ABS
15:
16:             .STACK  256                         ;スタック・セグメント
17:             .CODE                               ;コード・セグメント
18: start       PROC
19:             mov     ax, @data
20:             mov     ds, ax                      ;DSレジスタの初期化
21:             mov     ax, SEG data2
22:             push    ax                          ;data2のセグメント
23:             lea     ax, data2                   ;data2のセグメント
24:             push    ax                          ;data2へのポインタ
25:             push    data1                       ;ワード・データ
26:             call    sub1                        ;FARコール
27:             add     sp, 6
28:             push    data1     (data 2への       ;ワード・データ
29:             call    sub2       ポインタ)        ;NEARコール
30:             add     sp, 2
31:             mov     ah, 4Ch
32:             mov     al, 00h                     ;リターン・コード
33:             int     21h                         ;プログラム終了
34: start       ENDP
35:             END     start
```

〔リスト2-30〕外部参照のディレクティブの使用例(その2)

```
 1: ;*************************************************************
 2: ;
 3: ;   機      能 :   外部参照のプログラミング例その2
 4: ;   生      成 :   masm /ML extrn1;
 5: ;                  masm /ML extrn2;
 6: ;                  link /NOI/MAP extrn1 extrn2, extrn, extrn;
 7: ;
 8: ;*************************************************************
 9:             PAGE    ,132
10: data3       EQU     3333h
11:             .MODEL  SMALL, C                    ;メモリ・モデル/言語定義
12:
13:             PUBLIC  data1, data2, data3 ◄──(外部から参照可能にする)
14:             .DATA                               ;データ・セグメント
15: data1       DW      1111h
16: data2       DD      22226789h
17:             .CODE                               ;コード・セグメント
18:
19: sub1        PROC    FAR USES ax si di ds es, arg1:WORD, arg2:FAR PTR
20:             mov     ax, arg1                    ;data1
21:             lds     si, arg2                    ;data2へのポインタ
22:             les     di, [si]                    ;data2の内容
23:             ret
24: sub1        ENDP
25:
26: sub2        PROC    arg1
27:             mov     ax, arg1                    ;ABSデータ
28:             ret
29: sub2        ENDP
30:             END
```

■ データの定義と初期化

データ定義ディレクティブは，データ用のメモリ割り当てを宣言します．また，同時に割り当てられたデータ・エリアの初期化を指定することもできます．

データは，数値や文字列あるいは定数に評価される式として指定することができます．データの宣言にはDefineディレクティブを使用します．

Defineの構文

```
     |DB|
     |DW|
name |DD|initializer [,…]
     |DF|
     |DQ|
     |DT|
```

nameは，変数(データ)に割り当てるシンボル名でありオプショナルです．nameを指定しない場合は，変数用のメモリ割り当ては行われますが，その領域を名前でアクセスすることはできません．

このときに，オペランドにinitializer(初期値)を記述すると，メモリはそのinitializerで初期設定されます．ここで，initializerに演算子"?(疑問符)"を用いると，その変数はゼロで初期化されます．また，演算子DUPを用いることにより，大きなデータ領域(配列)を確保したり，データを繰り返して宣言することができます．演算子については後述します．

● データの構造化

データの構造化というのは，複数のデータをひとかたまりのデータ・ブロックとして扱えるようにしたもので，データの表現がわかりやすくなるというメリットがあり，大規模なプログラムにおいて複雑なデータを組織化するのに役立ちます．MS-DOSではストラクチャとレコードがあります．

◆ ストラクチャ

C言語のstruct宣言にみられるように，高級言語の構造体変数(ストラクチャ)をアセンブリ・プログラム・レベルでも実現可能としているのがSTRUCディレクティブです．STRUCディレクティブは，構造化したデータを扱いたい場合に，そのデータ・ブロックに名前をつけて一括したアクセスを可能にします．

MASMのSTRUCディレクティブでは，フィールドに定義されたデータ領域を初期化することが可能になっています．また，メモリ中の構造化されたデータをアクセスする際に，そのデータのベース・アドレスにそれぞれのフィールド名を添えて指定することで，フィールドの大きさにあったオペランドを生成することも可能になっています．

STRUCの構文

nameは，データ・ブロックの型に付けられる名前です．fielddeclarationには，そのデータ・ブロックのフィールド群(実際のデータ・アロケーション)をDefineディレクティブ(DB/DW/DDなど)を用いて定義します．ここで，フィールド群の個数に制限はありません．

filednameは，そのストラクチャを定義するソース・ファイルの中で他の変数名やラベル名と重複してはなりません．このfielddeclarationが定義された時点で，fieldnameはストラクチャの先頭から該当するフィールドまでのオフセットを表すようになります．

ここで重要なことは，STRUC～ENDSディレクティブでは，データの構造を定義したにすぎず，実際のメモリへの割り当てや初期化については，変数として宣言して別に行われなければなりません．ストラクチャを実際のメモリに割り当てるには，次の構文を用います．

ストラクチャのメモリへの割り当て

```
[name] structurename <[initialvalue [,…]>
```

nameは，変数に割り当てる名前です．nameはオプショナルであり，省略された場合は，その変数のためのスペースは割り当てられますが，変数に対して名前は与えられません．

structurenameは，それ以前にSTRUC～ENDSディレクティブを用いて定義した構造体型の名前です．構造体の各fieldについてinitialvalue(初期値)を指定することができます．

このinitialvalueの型は，対応するフィールドの型と合っていなければなりません．また，初期値をまったく指定しない場合でも，山形カッコ<>は必要になります．

ストラクチャ変数をアクセスするには，次の構文を用います．

```
variable.field
```

variableは，ストラクチャ変数の名前(またはストラクチャのアドレスに対応するオペランド)でなければなりません．fieldは，ストラクチャの中のフィールド名でなければなりません．variableとfiledの間は

".(ピリオド)" で区切ります．

MASM は，variable と field のオフセットの和を自動的に算出し，オペランドにこれらのオフセットの和を与えることにより，フィールドへのアクセスを可能にしています．

【STRUC のサンプル・プログラム】

リスト2-31(struc.lst)は，STRUC ディレクティブの使用例を示しています．同リストにおいて，シンボル strc1 や strc2 は，ストラクチャの名前であり，この名前で実際にメモリの割り当てが行われるわけではありません．実際のメモリ割り当ては，data1 や data2 のようにストラクチャの型をもつ変数として宣言することによって可能になります．

ここで，初期化の可能なフィールド(たとえば fild1 や fild3)に対して初期化データ('DATA1' や 1111222H)を指定することが可能です．もし，初期化の不可能なフィールド(たとえば fild2)に対して初期化データを指定すると MASM からエラーが返されます．

実際にメモリに割り当てられた各フィールドにアクセスするには，

　data1.fild1

のように記述します．また，ストラクチャ変数へのポインタをインデックス・レジスタに設定することにより，たとえば

　[di.memb2]

のように記述することによって，そのフィールド名をディスプレースメントとして使用することも可能です．

◆ レコード

ストラクチャがバイト単位でフィールドを定義するのに対し，レコードはビット単位でフィールドを定義することができます．レコードを使用すると，フィールド名からデータをフィールドに当てはめるために必要なビット・マスクやシフト数を求めることが可能になります．

レコードを使用するためには，まず RECORD ディレクティブを用いてレコードの型を定義します．レコードの型は，データのもととなる型であり，その定義はレコード内のビット・フィールドのサイズを決定するものです．レコード型は定義するときに初期化することも可能です．レコード型の定義だけでは，ストラクチャ型の定義と同様に，実際のメモリ割り当ては行われません．

レコードに対してメモリ割り当てを行うには，そのレコード型をもつ1個以上の変数の宣言を行います．このレコード変数の宣言によって，レコード型で定義されたフォーマットにしたがって実際のメモリの割り当てが行われます．

レコード変数は，レコード名を使って一括してアクセスできるほか，レコード名とフィールド名をフィールド演算子で組み合わせることによって，個別のフィールドをアクセスすることも可能です．

RECORD の構文

```
recordname RECORD field [,…]
```

recordname はレコード型の名前であり，レコードには最大16ビットまでのビット・フィールドを指定することができます．field には，フィールドの名前や幅および初期値などを定義します．それぞれの field の構文は次の書式にしたがいます．

フィールドの構文

```
fieldname:width [=expression]
```

fieldname はレコード内のフィールドの名前であり，他の変数名やラベル名と重複しない名前を用いる必要があります．width には，そのフィールドに含まれるビット数を指定し，1～16 ビットの範囲で指定しなければなりません．ビット数の合計が8ビット未満の場合，MASM はレコードに1バイトを用意し，それ以上の場合は2バイトを用意します．また，ビット数の合計が8ビットまたは16ビットに満たない場合は，フィールドをバイトまたはワードの中で右詰めにし，残りのビットには0を入れます．

RECORD ディレクティブによってレコードの型を定義しただけでは，実際のメモリ割り当ては行われません．レコードをメモリに対して割り当てるには，次のようにレコード変数の宣言を行います．

レコード変数の宣言

```
[name] recordname <[initialvalue [,…]]>
```

name はレコード変数の変数名となります．name はオプショナルであり，省略した場合は，その変数のためのメモリ割り当ては行われるものの，変数名は与えられないため変数名を用いてアクセスすることはできません．

recordname は，以前に RECORD ディレクティブを用いて定義したレコード型の名前です．レコードの各フィールドには，initialvalue(初期値)を指定することができます．この初期値は，整数や定数またはフィールドのサイズに合った式でなければなりません．初期値をまったく指定しない場合でも，山形カッコ<>は必要になります．

レコード変数をアクセスするにはレコード・オペランドを用います．レコード・オペランドの書式は次のようになっていて，これとレコード変数とを混同しないように注意が必要です．

```
recordname <[value [,…]]>
```

recordname は，ソース・ファイル中で定義されているレコード型の名前でなければなりません．value

〔リスト2-31〕STRUCディレクティブの使用例

```
                    ;*****************************************************************
                    ;   機    能：       STRUCディレクティブの使用例
                    ;   生    成：       masm /ML struc;
                    ;                    link /NOI/MAP struc,,struc;
                    ;
                    ;*****************************************************************
                                         PAGE   ,132
                                         .MODEL SMALL   ストラクチャ名       ;メモリ・モデル
                              strc1      STRUC ←        strc1の定義
 0000    20 20 20 20 20 20    fild1      DB                                  ;初期化可能
         20 20 20 20
   000A  0000 0000            fild2      DW      ?, ? ← 初期化できない      ;初期化不可能
   000E  00000000             fild3      DD      ?                           ;初期化可能
   0012                       strc1      ENDS

                              strc2      STRUC ←    ストラクチャ名strc2の定義
 0000    0008[                memb1      DB      8 DUP (?)                   ;初期化不可能
          ??
             ]                                   初期化できない

 0008    FFFF                 memb2      DW      0FFFFh                      ;初期化可能
 000A    00000000 00000000    memb3      DD      ?, ? ←                      ;初期化不可能
 0012                         strc2      ENDS

                                         .STACK 256                          ;スタック・セグメント
                                         .DATA                               ;データ・セグメント
 0000    44415441312020202020 data1      strc1 <'DATA1',,11112222h>          ;ストラクチャ
         20
 000A    0001[
          0000
          0000
             ]                                                 初期化データ
 000E    22221111

 0012    0008[                data2      strc2<,3333h>                       ;ストラクチャ
          ??
             ]
 001A    3333
 001C    0001[
          00000000
          00000000
             ]

 0024    0008[                data3      strc2 8 DUP (<,4444h>)              ;ストラクチャ配列
   0008[
      ??                                       ストラクチャ配列の確保
         ]
      4444
   0001[
      00000000
      00000000
         ]
            ]
                                         .CODE                               ;コード・セグメント
 0000                         start      PROC
 0000    B8 ---- R                       mov     ax, @data
 0003    8E D8                           mov     ds, ax                      ;DSレジスタの初期化
 0005    8D 1E 0000 R                    lea     bx, data1.fild1             ;fild1へのポインタ
 0009    C7 06 000A R 4444               mov     data1.fild2, 4444h          ;fild2の初期化
 000F    C7 06 000C R 5555               mov     data1.fild2 + 2, 5555h      ;fild2の初期化
 0015    C4 36 000E R                    les     si, data1.fild3             ;fild3のアクセス
 0019    1E                              push    ds
 001A    07                              pop     es                          ;ESレジスタの初期化
 001B    8D 3E 0012 R                    lea     di, data2.memb1             ;memb1へのポインタ
 001F    B9 0008                         mov     cx, SIZE memb1              ;memb1のサイズ
 0022    B0 FF                           mov     al, 0FFh
 0024    F3/ AA                          rep     stosb                       ;memb1の初期化
 0026    8D 3E 0012 R                    lea     di, data2                   ;data2へのポインタ
 002A    8B 45 08                        mov     ax, [di.memb2]              ;memb2のアクセス
 002D    C7 45 0A 6666                   mov     WORD PTR [di.memb3], 6666h  ;memb3へのアクセス
 0032    C7 45 0C 7777                   mov     WORD PTR [di.memb3 + 2], 7777h

 0037    B4 4C                           mov     ah, 4Ch
 0039    B0 00                           mov     al, 00h                     ;リターン・コード
 003B    CD 21                           int     21h                         ;プログラム終了
 003D                         start      ENDP
                                         END     start
```

はオプショナルであり，そのレコードの中のフィールドの値です．

【RECORDのサンプル・プログラム】

リスト2-32(record.lst)は，RECORDディレクティブの使用例を示しています．同リストのプロシージャget_timeは，引数として渡されたファイル・ハンドルとMS-DOSのファンクション・リクエスト57Hを用いて，ファイルの時間データを読み出し，その情報を各レジスタに設定して返すものです．

ファンクション・リクエスト57H(ファイルの時間データの読み出し)では，戻り値としてDXレジスタに日付データを，CXレジスタに時刻データを返し，それぞれのフォーマットは**図2-13**に示すように各レジスタのビット・フィールドに割り当てられています．

同リストでは，これらのビット・フィールドの割り当てを行うためにRECORDディレクティブを用いて，レコード型datedefおよびtimedefとして定義しています．

これらのレコードを実際にメモリに割り当てるには，変数dateやtimeのようにそのレコード名を用いて宣言し，これらの変数の各ビット・フィールドを初期化することも可能です．また，ビット・フィールドの取り出しは，該当するフィールド名とMASK演算子を用いて行います．

マクロ機能

マクロ機能とは，CPUニーモニックで書かれた一連の命令ブロックを一つの名前として参照できるようにした機能です．ユーザは，その際につけられた名前をソース・プログラムのオペレーション・フィールドに他のCPU命令ニーモニックと同様に記述することができます．このマクロ機能を用いることによって，たとえば拡張された命令セットをもつCPUを用いてプログラムするのと同様の機能を実現することが可能となります．また，マクロ機能はサブルーチン・コールとは異なり，MASMによりその名前が現れた段階で，そのロケーションに実際の命令ブロックが展開されていきます．

マクロ機能の利用形態としては，たとえばシステム・コールやユーザの作成した汎用サブルーチンなどを，あるファイルにマクロ定義しておきます．次に，これを必要とするソース・ファイルに，INCLUDEディレクティブを使ってそのルーチンを取り込むことによってマクロ展開を行うことができます．これによって，共通の処理を行うサブルーチンや関数などを，そのつどソース・ファイルにタイプ入力する手間が省けるばかりでなく，結果としてプログラムの保守性も一段と向上することになります．

また，MASMのマクロ機能では，定義されたマクロに対して入出力パラメータを渡すこともできます．したがって，これらのパラメータの入出力関係さえわかっていれば，そのマクロ内での処理に関してはブラック・ボックスとして扱うことができ，ユーザのプログラム管理の負担が軽減されます．

● マクロの定義

ユーザがマクロを使用するには，まえもってマクロ定義を行います．このため，マクロ定義は通常プログラムの最初に定義するか，マクロ定義の部分だけをあらかじめ別ファイルに作成しておき，INCLUDEディレクティブを用いてそのマクロ定義ファイルの取り込みを行うという方法が一般的です．

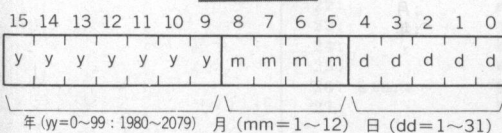
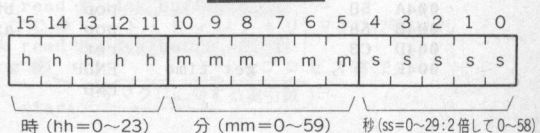
マクロ定義は，MACRO～ENDMディレクティブの間で行われ，その間の命令の集まりをマクロ・ボディと呼びます．

MACROの構文

```
name MACRO [parameter [,…]]
  ⋮       (macrobody)
name ENDM
```

nameはマクロ・ブロックに付けられる名前であり，マクロの呼び出しはこのnameを用いて行われます．parameterはオプショナルであり，マクロ・ブロック内で使用される実引数と1対1で置き換えられる仮の引数です．マクロの呼び出しは，マクロ定義の後であればプログラムの任意の場所で行うことができ，その構文は次のようになっています．

```
name [argument [,…]]
```

〔図2-13〕日付と時刻のビット・フォーマット

〔リスト2-32〕RECORDディレクティブの使用例

```
                         ;****************************************************************
                         ;
                         ;   機     能 :    RECORDディレクティブの使用例
                         ;   生     成 :    masm /ML record;
                         ;                  link /NOI/MAP record,,record;
                         ;
                         ;****************************************************************
                                  PAGE     ,132
                                  .MODEL   SMALL, C              ;メモリ・モデル／言語定義
                                  .STACK   256                   ;スタック・セグメント

  (レコード名)          datedef   RECORD   yyy:7, mmm:4, ddd:5   ;日付データの構造
                        timedef   RECORD   hhh:5, min:6, sss:5   ;時間データの構造
                                                                 (レコードの定義)
                                  .DATA                          ;データセグメント
0000   1241             date      datedef  <9, 2, 1>             ;1989年 2月 1日の設定
0002   53CF             time      timedef  <10, 30, 15>          ;10時 30分 30秒の設定
  (レコード変数の確保)                                            (レコードの初期化)
                                  .CODE                          ;コード・セグメント
0000              get_time        PROC     arg1:WORD
0003   8B 5E 04                   mov      bx, arg1              ;ファイル・ハンドル
0006   32 C0                      xor      al, al                ;読み出しモード
0008   B4 57                      mov      ah, 57h               ;ファンクション 57h
000A   CD 21                      int      21h                   ;ファンクション・リクエスト

000C   8B C1                      mov      ax, cx                ;AX ← 時間データ
                                  ;年データ
000E   8B DA                      mov      bx, dx                ;BX ← 日付データ
0010   81 E3 FE00                 and      bx, MASK yyy          ;yyy以外のフィールドのマスク
0014   B1 09                      mov      cl, yyy               ;CL ← シフト・カウント
0016   D3 EB                      shr      bx, cl                ;BXを右にシフトし yyy設定
0018   53                         push     bx                    ;年をプッシュ
                                                                 (ビットフィールドの取り出し)
                                  ;月データ
0019   8B DA                      mov      bx, dx                ;BX ← 日付データ
001B   81 E3 01E0                 and      bx, MASK mmm          ;mmm以外のフィールドのマスク
001F   B1 05                      mov      cl, mmm               ;CL ← シフト・カウント
0021   D3 EB                      shr      bx, cl                ;BXを右にシフトし mmm設定
0023   53                         push     bx                    ;月をプッシュ

                                  ;日データ
0024   8B DA                      mov      bx, dx                ;BX ← 日付データ
0026   83 E3 1F                   and      bx, MASK ddd          ;ddd以外のフィールドのマスク
0029   53                         push     bx                    ;日をプッシュ

                                  ;時データ
002A   8B D8                      mov      bx, ax                ;BX ← 時間データ
002C   81 E3 F800                 and      bx, MASK hhh          ;hhh以外のフィールドのマスク
0030   B1 0B                      mov      cl, hhh               ;CL ← シフト・カウント
0032   D3 EB                      shr      bx, cl                ;BXを右にシフトし hhh設定
0034   53                         push     bx                    ;時をプッシュ
                                  ;分データ
0035   8B D8                      mov      bx, ax                ;BX ← 時間データ
0037   81 E3 07E0                 and      bx, MASK min          ;min以外のフィールドのマスク
003B   B1 05                      mov      cl, min               ;CL ← シフト・カウント
003D   D3 EB                      shr      bx, cl                ;BXを右にシフトし min設定
003F   53                         push     bx                    ;分をプッシュ
                                  ;秒データ
0040   8B D8                      mov      bx, ax                ;BX ← 時間データ
0042   83 E3 1F                   and      bx, MASK sss          ;sss以外のフィールドのマスク
0045   53                         push     bx                    ;秒をプッシュ

0046   5F                         pop      di                    ;秒
0047   5E                         pop      si                    ;分
0048   5A                         pop      dx                    ;時
0049   59                         pop      cx                    ;日
004A   5B                         pop      bx                    ;月
004B   58                         pop      ax                    ;年
004D   C3                         ret
004E              get_time        ENDP
                                  END
```

マクロが呼び出されると，MASMは，そのマクロのマクロ・ボディを呼び出された位置に展開し，マクロ定義のparameter(仮引数)をマクロ呼び出しのargument(実引数)に置き換えます．

マクロ定義を再定義するには，単に新しくマクロを定義するだけで可能となります．また，マクロの削除を行うためにPURGEディレクティブが用意されています．

PURGE macroname [,…]

macronameには，以前にMACRO～ENDMディレクティブを用いて定義されたマクロ名を指定でき，複数のマクロを一度に削除することも可能です．MASMは，このディレクティブを検出するとmacronameで指定したマクロをメモリ上から削除します．

【MACROのサンプル・プログラム】

リスト2-33(sys.mac)は，マクロ定義の例を示しています．マクロabs_dsk_readでは，引数diskで指定されたドライブから指定されたセクタ(引数startおよびnで指定)を読み込み，引数bufferで指定されたバッファに格納します．ここでは，システム・コールのINT 25Hを用いてこの機能を実現しています．

また，マクロend_progでは，ファンクション・リクエスト4CHを用いて引数ret_codeによって指定された終了コードをもってプロセス終了を行います．

通常，これらのマクロ定義は機能別に分離して専用のファイルに格納しておきます．たとえば，システム・コールのマクロ定義は，一つのファイルにまとめて記述しておきます．

リスト2-34(macro.asm)は，定義されたマクロの呼

〔リスト3-33〕
マクロ定義の例

```
 1: ;*****************************************************************
 2: ;
 3: ;    機    能：    マクロ定義の例
 4: ;
 5: ;*****************************************************************
 6: abs_dsk_read       MACRO     disk, buffer, start, n
 7:                    mov       al, disk        ;ドライブ番号
 8:                    lea       bx, buffer      ;バッファ・アドレス
 9:                    mov       cx, n           ;セクタ数
10:                    mov       dx, start       ;開始セクタ
11:                    int       25h             ;ディスクの直接読み出し
12:                    ENDM
13:
14: end_prog           MACRO     ret_code
15:                    mov       al, ret_code    ;プロセス終了コード
16:                    mov       ah, 4Ch         ;プロセス終了
17:                    int       21h             ;ファンクション・リクエスト
18:                    ENDM
```


〔リスト2-34〕
マクロ呼び出しの例

```
 1: ;*****************************************************************
 2: ;
 3: ;    機    能：    マクロの使用例
 4: ;    ファイル：    sys.mac ･･･ マクロ定義ファイル
 5: ;    生    成：    masm /ML macro;
 6: ;                  link /NOI/MAP macro,,macro;
 7: ;
 8: ;*****************************************************************
 9:                 PAGE      ,132
10:                 .MODEL    SMALL             ;メモリ・モデル
11:                 .STACK    256               ;スタック・セグメント
12:                 .DATA                       ;データ・セグメント
13: dsk_buff1       DB        1024 DUP (?)      ;ディスク・バッファ
14: dsk_buff2       DB        1024 DUP (?)      ;ディスク・バッファ
15:
16:                 .CODE                       ;コード・セグメント
17:                 INCLUDE sys.mac  ← マクロ・ファイルの取り込み
18: start           PROC
19:                 mov       ax, @data
20:                 mov       ds, ax            ;DSレジスタの初期化
21:                 ;ドライブBのルート・ディレクトリ
22:                 abs_dsk_read 1, dsk_buff1, 5, 1  ┐
23:                 ;ドライブCのルート・ディレクトリ  ├ マクロの呼び出し
24:                 abs_dsk_read 2, dsk_buff2, 5, 1  ┘
25:                 end_prog 0                  ;プロセス終了
26: start           ENDP            ↑ マクロに対する実引数
27:                 END       start
```

び出し例を示しています．同リストのプログラムでは，**リスト2-33**で定義したマクロ・ファイル sys.mac をディレクティブ INCLUDE によって取り込んでいます．そして，マクロ abs_dsk_read に対して，あたかも C の関数を呼ぶかのごとくに必要なパラメータを記述してマクロの呼び出しを行っています．

このようにマクロを使用すると，ある機能の入出力パラメータさえ知っておけばよく，プログラムの記述が簡単かつ見通しのよいものとなります．このマクロの使用は，大きなプログラムを記述するうえで非常に有効な手段となります．

リスト2-35(macroasm.lst)は，MASM から出力さ

〔リスト2-35〕アセンブリ・リストの例

```
                ;***********************************************************
                ;
                ;     機   能 :    マクロの使用例
                ;     ファイル :    sys.mac ‥‥ マクロ定義ファイル
                ;     生   成 :    masm /ML macro;
                ;                  link /NOI/MAP macro,,macro;
                ;
                ;***********************************************************
                                PAGE        ,132
                                .MODEL      SMALL           ;メモリ・モデル
                                .STACK      256             ;スタック・セグメント
                                .DATA                       ;データ・セグメント
 0000  0400[    dsk_buff1       DB          1024 DUP (?)    ;ディスク・バッファ
       ??
           ]

 0400  0400[    dsk_buff2       DB          1024 DUP (?)    ;ディスク・バッファ
       ??
           ]
                                .CODE                       ;コード・セグメント
                                INCLUDE sys.mac
             C  ;***********************************************************
             C  ;
             C  ;     機   能 :    マクロ定義の例
             C  ;
             C  ;***********************************************************
             C  abs_dsk_read    MACRO      disk, buffer, start, n
             C                  mov        al, disk         ;ドライブ番号
             C                  lea        bx, buffer       ;バッファ・アドレス
             C                  mov        cx, n            ;セクタ数
             C                  mov        dx, start        ;開始セクタ
             C                  int        25h              ;ディスクの直接読み出し
             C                  ENDM
             C
             C  end_prog        MACRO      ret_code
             C                  mov        al, ret_code     ;プロセス終了コード
             C                  mov        ah, 4Ch          ;プロセス終了
             C                  int        21h              ;ファンクション・リクエスト
             C                  ENDM
             C
 0000                   start   PROC
 0000  B8 ---- R                mov        ax, @data
 0003  8E D8                    mov        ds, ax           ;DSレジスタの初期化
                                ;ドライブBのルート・ディレクトリ
                          abs_dsk_read 1, dsk_buff1, 5, 1
 0005  B0 01               1    mov        al, 1            ;ドライブ番号
 0007  8D 1E 0000 R        1    lea        bx, dsk_buff1    ;バッファ・アドレス
 000B  B9 0001             1    mov        cx, 1            ;セクタ数
 000E  BA 0005             1    mov        dx, 5            ;開始セクタ
 0011  CD 25               1    int        25h              ;ディスクの直接読み出し
                                ;ドライブCのルート・ディレクトリ
                          abs_dsk_read 2, dsk_buff2, 5, 1
 0013  B0 02               1    mov        al, 2            ;ドライブ番号
 0015  8D 1E 0400 R        1    lea        bx, dsk_buff2    ;バッファ・アドレス
 0019  B9 0001             1    mov        cx, 1            ;セクタ数
 001C  BA 0005             1    mov        dx, 5            ;開始セクタ
 001F  CD 25               1    int        25h              ;ディスクの直接読み出し
                          end_prog 0                        ;プロセス終了
 0021  B0 00               1    mov        al, 0            ;プロセス終了コード
 0023  B4 4C               1    mov        ah, 4Ch          ;プロセス終了
 0025  CD 21               1    int        21h              ;ファンクション・リクエスト
 0027                   start   ENDP
                                END        start
```

れたアセンブル・リストで，リスト出力に関するディレクティブを使用しない限り，取り込んだファイルの内容やマクロ展開のようすを知ることができます．

● リピート・ディレクティブ

マクロの特殊な形式として，繰り返しブロックの定義があります．繰り返しブロックがマクロ定義と異なる点は，繰り返しブロックには名前がつかないため呼び出しが不可能である点です．しかし，繰り返しブロックの内部では，マクロ定義と同様にパラメータを使用することが可能になっています．

◆ REPT

REPTディレクティブは，引数のない単純な繰り返しに用いられます．

REPTの構文

```
REPT expression
   ⋮
ENDM
```

MASMは，REPT～ENDMディレクティブの間にある定義ブロックをexpressionの回数だけ繰り返してアセンブルします．ここで，expressionは数値定数(符号なし16ビット)に評価される式でなければなりません．

◆ IRP

IRPディレクティブは，引数のリストを使って繰り返し回数と，それぞれの繰り返しにおけるパラメータを指定することによって繰り返しブロックの作成を行います．

IRPの構文

```
IRP parameter, <argument [,…]>
   ⋮
ENDM
```

このIRPディレクティブでは，山形カッコ<>で囲まれたリスト中の各argumentにつき，ブロック内部のステートメントが1回ずつ繰り返してアセンブルされます．parameterは，現在の引数で置き換えられる置換部分の名前です．引数としてはシンボルや文字列，あるいは数値定数などを指定することができます．

◆ IRPC

IRPCディレクティブは，ストリングを使って繰り返し回数とそれぞれの繰り返しにおけるパラメータを指定することによって，繰り返しブロックの作成を行います．

すなわち，IRPCディレクティブは，文字列を引数としてもち，その1文字ずつが操作の対象となる場合に使用します．

IRPCの構文

```
IRPC parameter,string
   ⋮
ENDM
```

ここで，parameterはstring中の現在の文字で置き換えられる置換部分の名前です．

【REPT，IRP，IRPCのサンプル・プログラム】

リスト2-36(rept.lst)は，繰り返しディレクティブの使用例で，同リストはMASMから出力されたアセンブル・リストを示しています．

同リストにおいて，REPTディレクティブの展開では，英大文字の数だけのバイトを確保し，その文字に対応したASCIIコードで初期化を行っています．

IRPCディレクティブの応用では，文字列に指定された各英大文字に対応した大文字および小文字，数値バージョンのASCIIコードを割り当ててバイトの確保を行っています．

また，IRPディレクティブの応用では，各レジスタのPUSH命令やPOP命令を自動生成しています．これは，PROCディレクティブによるUSESフィールドの指定と同等の機能を実現しているものです．

〔リスト2-36〕繰り返しディレクティブの使用例 ①

```
                ;*************************************************************
                ;
                ;     機    能：  繰り返しディレクティブの使用例
                ;     生    成：  masm /ML rept;
                ;                 link /NOI/MAP rept,,rept;
                ;
                ;*************************************************************
                          PAGE      ,132
                          .MODEL    SMALL                       ;メモリ・モデル
                          .STACK                                ;スタック・セグメント
                          .DATA                                 ;データ・セグメント

 0000           alpha     LABEL     BYTE
 = 0000         x         =         0                           ;カウンタ初期化
                          REPT      26                          ;26回の繰り返し
                          DB        'A' + x                     ;文字コードの宣言
                x         =         x + 1                       ;カウンタ・インクリメント
                          ENDM
```

〔リスト2-36〕繰り返しディレクティブの使用例 ②

```
 0000  41          1    DB      'A' + x            ;文字コードの宣言
 0001  42          1    DB      'A' + x            ;文字コードの宣言
 0002  43          1    DB      'A' + x            ;文字コードの宣言
 0003  44          1    DB      'A' + x            ;文字コードの宣言
 0004  45          1    DB      'A' + x            ;文字コードの宣言
 0005  46          1    DB      'A' + x            ;文字コードの宣言
 0006  47          1    DB      'A' + x            ;文字コードの宣言
 0007  48          1    DB      'A' + x            ;文字コードの宣言
 0008  49          1    DB      'A' + x            ;文字コードの宣言
 0009  4A          1    DB      'A' + x            ;文字コードの宣言
 000A  4B          1    DB      'A' + x            ;文字コードの宣言
 000B  4C          1    DB      'A' + x            ;文字コードの宣言
 000C  4D          1    DB      'A' + x            ;文字コードの宣言
 000D  4E          1    DB      'A' + x            ;文字コードの宣言
 000E  4F          1    DB      'A' + x            ;文字コードの宣言
 000F  50          1    DB      'A' + x            ;文字コードの宣言
 0010  51          1    DB      'A' + x            ;文字コードの宣言
 0011  52          1    DB      'A' + x            ;文字コードの宣言
 0012  53          1    DB      'A' + x            ;文字コードの宣言
 0013  54          1    DB      'A' + x            ;文字コードの宣言
 0014  55          1    DB      'A' + x            ;文字コードの宣言
 0015  56          1    DB      'A' + x            ;文字コードの宣言
 0016  57          1    DB      'A' + x            ;文字コードの宣言
 0017  58          1    DB      'A' + x            ;文字コードの宣言
 0018  59          1    DB      'A' + x            ;文字コードの宣言
 0019  5A          1    DB      'A' + x            ;文字コードの宣言
                        IRPC    char, XYZABC
                        DB      '&char'            ;数値
                        DB      '&char' + ' '      ;大文字
                        DB      '&char' - '@'      ;小文字
                        ENDM
 001A  58          1    DB      'X'                ;数値
 001B  78          1    DB      'X' + ' '          ;大文字
 001C  18          1    DB      'X' - '@'          ;小文字
 001D  59          1    DB      'Y'                ;数値
 001E  79          1    DB      'Y' + ' '          ;大文字
 001F  19          1    DB      'Y' - '@'          ;小文字
 0020  5A          1    DB      'Z'                ;数値
 0021  7A          1    DB      'Z' + ' '          ;大文字
 0022  1A          1    DB      'Z' - '@'          ;小文字
 0023  41          1    DB      'A'                ;数値
 0024  61          1    DB      'A' + ' '          ;大文字
 0025  01          1    DB      'A' - '@'          ;小文字
 0026  42          1    DB      'B'                ;数値
 0027  62          1    DB      'B' + ' '          ;大文字
 0028  02          1    DB      'B' - '@'          ;小文字
 0029  43          1    DB      'C'                ;数値
 002A  63          1    DB      'C' + ' '          ;大文字
 002B  03          1    DB      'C' - '@'          ;小文字
                        .CODE                      ;コード・セグメント
 0000           sub1    PROC
                        IRP     reg, <ax, bx, cx, dx>
                        push    reg                ;PUSH命令の生成
                        ENDM
 0000  50          1    push    ax                 ;PUSH命令の生成
 0001  53          1    push    bx                 ;PUSH命令の生成
 0002  51          1    push    cx                 ;PUSH命令の生成
 0003  52          1    push    dx                 ;PUSH命令の生成

                        IRP     reg, <dx, cx, bx, ax>
                        pop     reg                ;POP命令の生成
                        ENDM
 0004  5A          1    pop     dx                 ;POP命令の生成
 0005  59          1    pop     cx                 ;POP命令の生成
 0006  5B          1    pop     bx                 ;POP命令の生成
 0007  58          1    pop     ax                 ;POP命令の生成
 0008           sub1    ENDP
                        END
```

MASMにより自動生成される

IRPCディレクティブの使用例

MASMにより自動生成される

IRPディレクティブの使用例

MASMにより自動生成される

条件アセンブル

条件アセンブルとは，アセンブル中にある条件を判断して，その命令をアセンブルするかどうかをMASMに指定するものです．これにより，不必要な部分のアセンブルが省略され，オブジェクト・コードがコンパクトになります．

また，ある類似したプログラムで，その処理内容が一部だけ異なるような場合に，この条件アセンブルを指定すると，ソース・プログラムは1本あればよく，これもプログラムの保守性の向上につながることになります．

ただし，これらの条件は，プログラムの実行中のレジスタやフラグの内容には無関係で，あくまでもアセンブル中に得られる情報に限定されることに注意が必要です．

MASMでは，条件アセンブルを実現するために豊富な条件ディレクティブが用意されています．条件ディレクティブは，一般に次の構文で記述します．

条件ディレクティブの構文

```
IFxxxx condition
  :        (ifstatements)
  :
[ELSE
  :        (elsestatements)
  :  ]
ENDIF
```

条件ディレクティブは，必ずIFxxxxディレクティブで始まりENDIFディレクティブで終了します．conditionは，条件ディレクティブの条件であり，MASMはこのconditionの評価を行い，その結果が真(非ゼロ)であれば，IFxxxxディレクティブに続くブロック(ifstatements)をアセンブルします．また，conditionが偽(ゼロ)であれば，ELSEディレクティブに続くブロック(elsestatements)をアセンブルします．

ELSEディレクティブおよびelsestatementsは省略できますが，ENDIFディレクティブは省略できません．

もし，ELSEブロックが省略されている場合は，conditionが真のときにIFxxxx～ENDIFのブロック(ifstatements)がアセンブルされ，conditionが偽の場合はなにもアセンブルされません．条件ディレクティブは255レベルまでネストすることが可能です．

【条件アセンブルのサンプル・プログラム】

リスト2-37(if.asm)は条件アセンブルの使用例を示しています．**リスト2-38**(if.mac)は，**リスト2-37**のINCLUDEディレクティブによって取り込まれるマクロ定義のファイル内容で，64ビット加算ルーチンadd_64と32ビット加算ルーチンadd_32が記述されています．

リスト2-37において，最初のIFディレクティブは，data1が4バイトで定義されていれば32ビット加算ルーチンのマクロを展開し，もしdata1が8バイトで定義されている場合には64ビット加算ルーチンのマクロ展開を行います．

以下data3，data5についてもIFディレクティブによって，データの長さに合った加算ルーチンの展開を行います．ここで，data5は16ビット長であるため何の命令コードも展開されないはずです．

リスト2-39(if.lst)は，MASMから出力されたアセンブル・リストです．同リストが示すように，IFディレクティブによってデータ長に合ったマクロが展開されていることが確認できます．また，data5に対してはデータ長が合っていないため何の命令コードも展開されていません．

● 8086 vs 68000(その1) レジスタ ●

8086では，16ビットの汎用レジスタが4本用意されています．このほかにユーザの使用できるポインタ・レジスタとして3本の16ビット・レジスタがあります．

これらのレジスタは，特殊な用途が限定されています．たとえばBXレジスタはポインタとして利用できますが，AXレジスタはポインタに利用することはできません．また，I/OのポインタにはDXレジスタしか利用できません．SIレジスタはソース側に，DIレジスタはデスティネーション側にデフォルトで指定されます．

一方，68000の場合は，汎用レジスタとして32ビット・レジスタ8本，ポインタ・レジスタとしてやはり8本の32ビット・レジスタが用意されています．汎用レジスタは，どれも同じ機能をもち，ポインタ・レジスタとして利用することもできます．強いて特殊なレジスタといえば，ポインタ・レジスタa7がシステム・スタックをポイントしていることくらいでしょう．

［リスト2-37］ 条件アセンブルの例

```
 1: ;*********************************************************
 2: ;
 3: ;  機      能 :  条件ディレクティブの使用例
 4: ;  生      成 :  masm /ML if;
 5: ;                link /NOI/MAP if,,if;
 6: ;
 7: ;*********************************************************
 8:            PAGE    ,132
 9:            .LFCOND                          ;偽の部分もリスト出力
10:            .MODEL  SMALL                    ;メモリ・モデル
11:            .DATA                            ;データ・セグメント
12: data1      DQ      1                        ;64ビット・データ
13: data2      DQ      2                        ;64ビット・データ
14:
15: data3      DD      3                        ;32ビット・データ
16: data4      DD      4                        ;32ビット・データ
17:
18: data5      DW      5                        ;16ビット・データ
19: data6      DW      6                        ;16ビット・データ
20:
21:            .CODE                            ;コード・セグメント
22:            INCLUDE if.mac ◀ マクロ定義ファイルの取り込み
23:
24: add        PROC
25:            mov     ax, @data
26:            mov     ds, ax                   ;DSレジスタの初期化
27:
28:            IF      SIZE data1 EQ 4
29:            add_32  data1, data2             ;32ビット加算
30:            ELSEIF  SIZE data1 EQ 8
31:            add_64  data1, data2             ;64ビット加算
32:            ENDIF
33:                                    もしdata1が4バイトであれば32ビット加算
34:            IF      SIZE data3 EQ 4
35:            add_32  data3, data4             ;32ビット加算
36:            ELSEIF  SIZE data3 EQ 8
37:            add_64  data3, data4             ;64ビット加算
38:            ENDIF
39:                                    もしdata1が8バイトであれば64ビット加算
40:            IF      SIZE data5 EQ 4
41:            add_32  data5, data6             ;32ビット加算
42:            ELSEIF  SIZE data5 EQ 8
43:            add_64  data5, data6             ;64ビット加算
44:            ENDIF
45:
46:            mov     ah, 4Ch
47:            mov     al, 00h                  ;リターン・コード
48:            int     21h                      ;プログラム終了
49: add        ENDP
50:            END
```

［リスト2-38］
if.mac ①

```
 1: ;*********************************************************
 2: ;
 3: ;  機      能:   マクロ定義の例
 4: ;
 5: ;*********************************************************
 6: add_32     MACRO   src_32, des_32           ;32ビット加算
 7:            lea     bx, des_32
 8:            lea     bp, src_32
 9:            mov     cx, 2
10:            mov     si, 0
11: do_32:
12:            mov     ax, ds:[bp+si]
13:            adc     [bx+si], ax
14:            add     si, 2
15:            loop    do_32
16:            ENDM
17:
18: add_64     MACRO   src_64, des_64           ;64ビット加算
19:            lea     bx, des_64
```

〔リスト2-38〕
if.mac ②

```
    20:             lea     bp, src_64
    21:             mov     cx, 4
    22:             mov     si, 0
    23: do_64:
    24:             mov     ax, ds:[bp+si]
    25:             adc     [bx+si], ax
    26:             add     si, 2
    27:             loop    do_64
    28:             ENDM
```

〔リスト2-39〕MASMから出力されたアセンブル・リスト ①

```
                          ;*****************************************************************
                          ;
                          ;    機    能：  条件ディレクティブの使用例
                          ;    生    成：  masm /ML if;
                          ;                link /NOI/MAP if,,if;
                          ;
                          ;*****************************************************************
                                     PAGE    ,132
                                     .LFCOND                          ;偽の部分もリスト出力
                                     .MODEL  SMALL                    ;メモリ・モデル
                                     .DATA                            ;データ・セグメント
0000  0100000000000000 data1         DQ      1                        ;64ビット・データ
0008  0200000000000000 data2         DQ      2                        ;64ビット・データ

0010  00000003         data3         DD      3                        ;32ビット・データ
0014  00000004         data4         DD      4                        ;32ビット・データ

0018  0005             data5         DW      5                        ;16ビット・データ
001A  0006             data6         DW      6                        ;16ビット・データ

                                     .CODE                            ;コード・セグメント
                                     INCLUDE if.mac
                       C  ;********************************************************************
                       C  ;
                       C  ;    機    能：  マクロ定義の例
                       C  ;
                       C  ;********************************************************************
                       C
                       C  add_32     MACRO   src_32, des_32            ;32ビット加算
                       C             lea     bx, des_32
                       C             lea     bp, src_32
                       C             mov     cx, 2
                       C             mov     si, 0
                       C  do_32:
                       C             mov     ax, ds:[bp+si]
                       C             adc     [bx+si], ax
                       C             add     si, 2
                       C             loop    do_32
                       C             ENDM
                       C
                       C  add_64     MACRO   src_64, des_64            ;64ビット加算
                       C             lea     bx, des_64
                       C             lea     bp, src_64
                       C             mov     cx, 4
                       C             mov     si, 0
                       C  do_64:
                       C             mov     ax, ds:[bp+si]
                       C             adc     [bx+si], ax
                       C             add     si, 2
                       C             loop    do_64
                       C
                       C             ENDM
                       C
0000                      add        PROC
0000  B8 ---- R                      mov     ax, @data
0003  8E D8                          mov     ds, ax                   ;DSレジスタの初期化
```

〔リスト2-39〕MASMから出力されたアセンブル・リスト ②

```
                              IF        SIZE data1 EQ 4
                              add_32    data1, data2          ;32ビット加算
                              ELSEIF    SIZE data1 EQ 8
                              add_64    data1, data2          ;64ビット加算
0005  8D 1E 0008 R    1            lea       bx, data2
0009  8D 2E 0000 R    1            lea       bp, data1
000D  B9 0004         1            mov       cx, 4
0010  BE 0000         1            mov       si, 0
0013                  1       do_64:
0013  3E: 8B 02       1            mov       ax, ds:[bp+si]
0016  11 00           1            adc       [bx+si], ax
0018  83 C6 02        1            add       si, 2
001B  E2 F6           1            loop      do_64
                              ENDIF

                              IF        SIZE data3 EQ 4
                              add_32    data3, data4          ;32ビット加算
001D  8D 1E 0014 R    1            lea       bx, data4
0021  8D 2E 0010 R    1            lea       bp, data3
0025  B9 0002         1            mov       cx, 2
0028  BE 0000         1            mov       si, 0
002B                  1       do_32:
002B  3E: 8B 02       1            mov       ax, ds:[bp+si]
002E  11 00           1            adc       [bx+si], ax
0030  83 C6 02        1            add       si, 2
0033  E2 F6           1            loop      do_32
                              ELSEIF    SIZE data3 EQ 8
                              add_64    data3, data4          ;64ビット加算
                              ENDIF

                              IF        SIZE data5 EQ 4
                              add_32    data5, data6          ;32ビット加算
                              ELSEIF    SIZE data5 EQ 8
                              add_64    data5, data6          ;64ビット加算
                              ENDIF

0035  B4 4C                   mov       ah, 4Ch
0037  B0 00                   mov       al, 00h               ;リターン・コード
0039  CD 21                   int       21h                   ;プログラム終了
003B                 add      ENDP
                              END
```


演算子

各ディレクティブやCPUニーモニックは，特定の型のオペランドを必要とします．大部分のディレクティブでは，文字列または数値定数，あるいはそれらの定数に評価されるシンボル，または式をオペランドとしてとります．

MASMは，これらオペランドを結合したり比較したり，変更あるいは解析するために，**表2-6**に示したようなさまざまな演算子を用意しています．

演算子は，数値演算子や関係演算子，型演算子など，いくつかのタイプに分類されます．

● セグメント演算子

ラベルや変数は，NEAR/WORDなどといった型のほかに，そのラベルや変数が定義された論理セグメントと，その論理セグメント内のオフセットなどを属性としてもっています．

OFFSET演算子やSEG演算子を用いると，これらのセグメントやオフセットなどのアドレスに関する属性の一部を取り出すことが可能になります．

◆ セグメント・オーバライド

セグメント・オーバライド演算子“：”を用いると，変数やラベルのアドレスを強制的に特定のセグメント相対で計算することができます．

：の構文

```
segment:expression
```

セグメント・オーバライド演算子は，ほかのセグメントにある変数やラベルなどのアクセスを可能にします．

〔表2-6〕MASMの演算子

分　類	演算子および構文	機　　能
算術演算子	expression1 + expression2	expression1とexpression2を加算する
	expression1 − expression2	expression1からexpression2を減算する
	expression1 * expression2	expression1とexpression2を乗算する
	expression1 / expression2	expression1をexpression2で除算する
	−expression	expressionの符号を反転する
	expression1 MOD expression2	expression1をexpression2で除算し，その剰余を返す
	variable.field	fieldのオフセットにvariableのオフセットを加えた値を返す
	[expression1] [expression2]	expression1のオフセットにexpression2のオフセットを加えた値を返す
マクロ演算子	<text>	マクロ引数内のtextを一つのリテラルな要素(文字列そのもの)として扱う
	! character	マクロ引数内のcharacterを一つのリテラルな要素(文字そのもの)として扱う
	;; text	マクロ展開時にtextをリストに出力されないコメントとして扱う
	% text	マクロ引数内のtextを式として扱う
	& parameter	parameterを対応する引数の値で置き換える
論理演算子とシフト演算子	expression1 AND expression2	expression1とexpression2に対し，ビット単位で論理積(AND)をとる
	expression1 OR expression2	expression1とexpression2に対し，ビット単位で論理和(OR)をとる
	expression1 XOR expression2	expression1とexpression2に対し，ビット単位で排他的論理和(EXOR)をとる
	NOT expression	expressionの全ビットの反転を行う
	expression SHL count	count回だけexpressionのビットを左にシフトする
	expression SHR count	count回だけexpressionのビットを右にシフトする
レコード演算子	MASK {recordfieldname|record}	recordfieldnameやrecordのビットをセットし，他のビットをクリアしたビット・マスク値を返す
	WIDTH {recordfieldname|record}	recordfieldnameまたはrecordの幅をビット数で返す
型演算子	HIGH expression	expressionの上位バイトを返す
	LOW expression	expressionの下位バイトを返す
	type PTR expression	expressionを強制的にtypeの型をもつものとして扱う
	SHORT label	labelの型をSHORT(128バイト以内)に指定する
	SIZE variable	variableがDUP演算子で定義されている場合に，variableに割り当てられているバイト数を返す
	THIS type	指定されたtypeのオセット値とセグメント値をもつオペランドを返す
	TYPE expression	expressionのバイト数やラベルの属性(NEAR|FAR)の型を返す
	.TYPE expression	expressionのモードや有効範囲を表すバイトを返す
セグメント	segment : expression	アドレスに対するデフォルトのセグメント・レジスタを変更する．segmentにはセグメント・レジスタやグループ名，expressionには定数，メモリ式，SEG式を指定できる
	SEG expression	expressionの定義されているセグメントを返す
	OFFSET expression	expressionの定義されているオフセットを返す
関係演算子	expression1 EQ expression2	expression1とexpression2を比較し，expression1とexpression2が等しければ真(−1)を返す．もし，そうでなければ偽(0)を返す
	expression1 NE expression2	expression1とexpression2を比較し，expression1とexpression2が等しくなければ真(−1)を返す．もし，そうでなければ偽(0)を返す
	expression1 GT expression2	expression1とexpression2を比較し，expression1がexpression2よりも大きければ真(−1)を返す．もし，そうでなければ偽(0)を返す
	expression1 GE expression2	expression1とexpression2を比較し，expression1がexpression2以上の場合に真(−1)を返す．もし，そうでなければ偽(0)を返す
	expression1 LT expression2	expression1とexpression2を比較し，expression1がexpression2よりも小さい場合に真(−1)を返す．もし，そうでなければ偽(0)を返す
	expression1 LE expression2	expression1とexpression2を比較し，expression1がexpression2以下の場合に真(−1)を返す．もし，そうでなければ偽(0)を返す
その他の演算子	; text	textをコメントとして扱う
	count DUP (initialvalue [...])	count個のinitialvalueの宣言を行う
	¥	論理行を次の物理行に継続する場合に行末に付加する

segmentには，セグメント・レジスタの名前(CSやSSなど)や論理セグメント名およびグループ名を指定することができます．

expressionには，定数や式またはSEG式を使用することが可能です．

このセグメント変更演算子は，たとえばCSセグメントやSSセグメント内にあるデータをアクセスしたい場合などに使用します．

◆ SEG

あるセグメントに定義されているシンボルのセグメント・アドレスを知りたい場合はSEG演算子を用います．

SEGの構文

```
SEG expression
```

expressionには，任意のラベルや変数名あるいはセグメント名またはグループ名などを指定することができます．SEG演算子は，たとえば変数の定義されているセグメントの絶対アドレスを，セグメント・レジスタに設定したいような場合に用います．

◆ OFFSET

SEG演算子に対して，あるラベルや変数のオフセット値を得るにはOFFSET演算子を用います．

OFFSETの構文

```
OFFSET expression
```

expressionには，任意のラベルや変数名またはその他のメモリ・オペランドを直接指定することができます．このOFFSET演算子は，レジスタ間接アドレッシングなどで，変数やラベルをアクセスしたい場合などによく用いられます．

● 型演算子

型演算子は，オペランドおよびその他の式の型を指定したり，強制的に変更したり，解析したりします．

◆ PTR

PTR演算子は，たとえばワードで定義されているデータに対してバイト単位でアクセスしたいような場合に，オペランドの型を強制的に変更するために使用します．

また，レジスタ間接アドレッシング命令などにおいて，それだけではMASMが型を判断できないような場合に，そのオペランドの型を明確に指定してやる場合にも使用されます．

PTRの構文

```
type PTR expression
```

PTR演算子は，オペランド(expression)の型に対して新たな型(type)を指定します．ただし，与えられたアドレス式(expression)自身の型(属性)を変更するものではありません．

〔表2-7〕TYPE演算子の返す値

expression		返す値
数値		0
変数	DB	1
	DW	2
	DD	4
	DF	6
	DQ	8
	DT	10
	STRUC	STRUCディレクティブによって定義されたバイト数
ラベル	NEAR	FFFFH
	FAR	FFFEH

◆ TYPE

TYPE演算子は，式で型を表す数値を返します．

TYPEの構文

```
TYPE expression
```

TYPE演算子では，表2-7に示したようにexpressionが変数に評価される場合には，その変数の中のデータのバイト数を返します．また，expressionがラベルの場合には，NEARに対してはFFFFHを，FARに対してはFFFEHを返します．

◆ .TYPE

.TYPE演算子は，式に外部参照が含まれているかどうか，LOCALディレクティブで局所的に宣言されたものかどうか，ラベル/変数名/定数の何に関係するかといった特性を知るために用いられます．

.TYPEの構文

```
.TYPE expression
```

expressionには，任意の式を使用することができますが，expressionが無効な場合にはゼロを返します．expressionが有効な場合には図2-14に示した値が返されます．

◆ LENGTH

LENGTH演算子は，DUP演算子で定義した変数のデータの個数を返します．ここで，返されるデータの個数は，変数のタイプ(バイト数)には依存しません．

LENGTHの構文

```
LENGTH variable
```

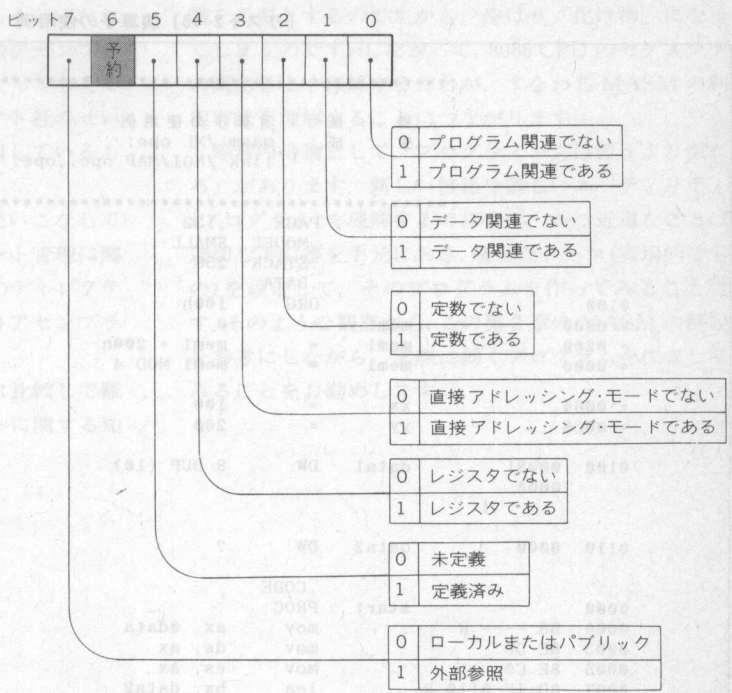
variableには，DUP演算子などを用いて定義されている変数の変数名を指定します．

◆ SIZE

SIZE演算子は，DUP演算子などを用いて定義した配列などの変数に割り当てられている総バイト数を返します．返される値は，LENGTH演算子の返す値とTYPE演算子の返す値の積に等しい値となります．

SIZEの構文

```
SIZE variable
```

〔図2-14〕
.TYPE 演算子で返される
ビットの意味

variable には，DUP 演算子などを用いて定義されている変数の変数名を指定します．

● 数値演算子

数値演算子には，算術演算子のほかにシフトやビット処理を行うために，シフト演算子や論理演算子などが用意されています．

算術演算子には，単項演算子である符号の“＋”と“－”があり，このほかに二項演算子が用意されています．単項演算子の＋は無視され，－は符号を変えるために使用されます．二項演算子には，一般の算術機能（加算，減算，乗算，除算，剰余）を行う演算子が用意されています．算術演算子はオペランドを結合して，結果としてデータ項目またはアドレスとなる式を作り上げるのに使用されます．

論理演算子は，二つのオペランドの対応するビット位置のバイナリ値を比較して，この演算子により定義される論理関係についての評価を行います．

シフト演算子は，指定された式のビットを右や左に指定されたビット数だけシフトします．

● 関係演算子

関係演算子は二項演算子であり，二つの定数オペランドの比較を行います．関係演算子では，二つのオペランドの関係が演算子と一致している場合にはFFFFH が返されます．また，一致していない場合はゼロが返されます．

〔表2-8〕演算子の優先順位

優先順位	演算子
高い	LENGTH, SIZE, WIDTH, MASK, ( ), [ ], < >
↑	.（構造体フィールド名）
	:（セグメント・オーバライド）
	PTR, OFFSET, SEG, TYPE, THIS
	HIGH, LOW
優先度	＋，－（単項演算子）
	*，/，MOD, SHL, SHR
	＋，－（2項演算子）
	EQ, NE, LT, LE, GT, GE
	NOT
	AND
↓	OR, XOR
低い	SHORT, .TYPE

● 演算子の評価順位

演算子の評価の順位は表2-8 にしたがって評価されます．同じ優先順位をもつ演算子は左から右へ評価されていきます．優先順位は，一般の式と同様にカッコでくくることによってユーザが強制的に変更することが可能です．

〔リスト2-40〕演算子の使用例

```
                        ;**********************************************************
                        ;
                        ;   機   能:   演算子の使用例
                        ;   生   成:   masm /ML ope;
                        ;              link /NOI/MAP ope,,ope;
                        ;
                        ;**********************************************************
                                PAGE    ,132
                                .MODEL  SMALL                   ;メモリ・モデル
                                .STACK  256                     ;スタック・セグメント
                                .DATA                           ;データ・セグメント
 0100                           ORG     100h
 = 0000                 mem1    =       0
 = 0200                 mem1    =       mem1 + 200h             ;算術演算子
 = 0000                 mem1    =       mem1 MOD 4              ;算術演算子

 = 0064                 xx      =       100
 = 00C8                 yy      =       200

 0100  0008[            data1   DW      8 DUP (10)
        000A
              ]

 0110  0000             data2   DW      ?

                                .CODE                           ;コード・セグメント
 0000                   start   PROC
 0000  B8 ---- R                mov     ax, @data
 0003  8E D8                    mov     ds, ax                  ;DSレジスタの初期化
 0005  8E C0                    mov     es, ax
 0007  8D 1E 0110 R             lea     bx, data2

 000B  B8 ---- R                mov     ax, SEG data1           ;セグメント演算子
 000E  B8 0100 R                mov     ax, OFFSET data1        ;セグメント演算子

 0011  B8 0002                  mov     ax, TYPE data1          ;型演算子
 0014  B8 0008                  mov     ax, LENGTH data1        ;型演算子
 0017  B8 0010                  mov     ax, SIZE data1          ;型演算子

 001A  89 07                    mov     [bx], ax                ;インデックス演算子
 001C  26: 89 07                mov     es:[bx], ax             ;セグメント変更演算子

 001F  B8 02E8                  mov     ax, 01011101B SHL 3     ;シフト演算子
 0022  B8 0011                  mov     ax, 01011101B AND 10110011B ;ビット論理演算子
 0025  B8 FF67                  mov     ax, NOT 10011000B       ;ビット論理演算子
 0028  B8 0000                  mov     ax, xx GE yy            ;関係演算子
 002B                   label1:
 002B  A0 0110 R                mov     al, BYTE PTR data2      ;型演算子
 002E  9A 003B ---- R           call    FAR PTR sub1            ;型演算子
 0033  EB F6                    jmp     SHORT label1            ;型演算子
 0035  B4 4C                    mov     ah, 4Ch
 0037  B0 00                    mov     al, 00h                 ;リターン・コード
 0039  CD 21                    int     21h                     ;プログラム終了
 003B                   start   ENDP

 003B                   sub1    PROC    FAR
 003B  CB                       ret
 003C                   sub1    ENDP
                                END     start
```

【演算子のサンプル・プログラム】

リスト2-40は演算子の使用例を示しています．

*　　　　　　　　　　*

MS-DOS上でプログラム開発を行う際に，C言語などの高級言語のみによってもある程度のプログラムは作成できます．しかし，本書の以降の章でも示しているように，場合によってはアセンブリ言語で記述したほうがプログラムの見通しがよい場合もあり，また，処理速度を上げたり，ウラ技的なプログラム開発を行う場合にはアセンブリ言語の使用も必須のものとなります．

MS-DOSでアセンブリ言語のプログラムを開発する場合，マイクロソフト社以外からもいくつかのアセンブラが市販されていますが，やはりマクロ・アセンブラMASMの利用方法は基本となるものであり，このMASMについては完全にマスタしておくべきです．

マクロ・アセンブラMASMは難解であるといわれています．筆者も使いはじめの頃は「このアセンブラは化け物だ」と思ったほどでした．今になって考えてみると，MASMの難しさはマイクロソフト社のせいではなく，8086 CPUのセグメントに起因しているものだということがわかってきました．

MASMのディレクティブと演算子を使いこなしていくと，難しく感じられた部分はセグメント管理に関するディレクティブと演算子であり，他のディレクティブや演算子は一般的な(他のCPU用の)アセンブラと大差ないことがわかります．

そもそもアセンブリ言語は，高級言語に比較して難しいものなのに，それに加えてセグメントに関する知識を必要とするのですから，やはり「化け物」になってしまうのです．したがって，8086 CPUのセグメントの概念をよく理解することが，すなわちMASMの利用方法を理解することにつながります．

筆者の持論として，「プログラミングは習うより慣れろ」があります．新しい言語や難しいユーティリティ(プログラム)を理解するのに必要，かつ近道なことは，適切な手引書を手元におき，適当なテーマ(実用的なもの)を設定して，そのプログラムを作ってみることです．そのような観点から，この第2章のMASMの解説を参考にしながら，実際に動くプログラムを作成してみることをお勧めします．

第3章 MS-DOSの内部構造

ブートストラップとプロセスとメモリ・モデル

本章では，MS-DOSの内部構造について解説していきます．

まず，MS-DOSのメモリ配置を知る意味から，システムのブート手順について解説します．次に，外部コマンドやユーザの作成したプログラムが，どのようにメモリに配置され，どのような手順で実行されるのかを解説してあります．また，これにともないプログラムの実行環境やメモリ・モデルなどについても言及して，ユーザがプログラムを作成する際のルールについても解説していきます．

3-1 MS-DOSのブート

MS-DOSでは，システム用のファイルを次のように三つのファイルに分割しています．**図3-1**に示すように，それぞれのファイルが各機能を分担してMS-DOSとして成り立っています．

io.sys　　　…入出力制御
msdos.sys　 …ファイルおよびメモリ管理
command.com …コマンド実行

このように，各機能をそれぞれのモジュールに分割するのにはそれなりの理由があります．それは，これらのDOSは多くのシステム(ハードウェア)に移植されるため，さまざまなシステムをサポートする必要があるからです．

そこでMS-DOSでは，まずio.sysをmsdos.sysと別のモジュールにすることにより，各機種への移植性を高めています．io.sysは，ディスク・ドライバやコンソール入出力など，標準デバイスのドライバの集合で，ハードウェアに密着した部分であり，機種への依存度が極めて高い部分です．

したがって，このio.sysを別のモジュール(ファイル)にしておけば，OEMメーカ側では，このio.sysだけをその機種に応じて開発すればよく，msdos.sysやcommand.comの移植作業は不要となるため，移植の際の手間がかなり省略できることになります．

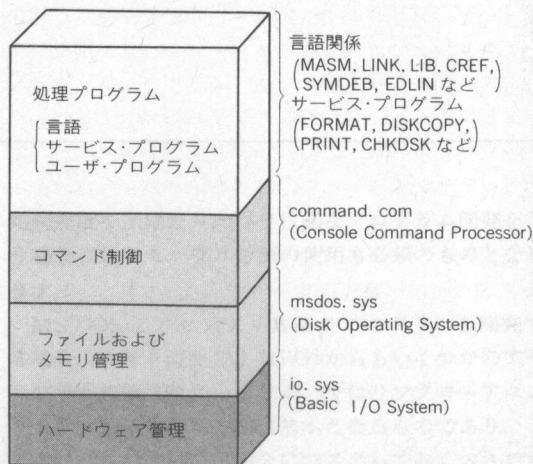
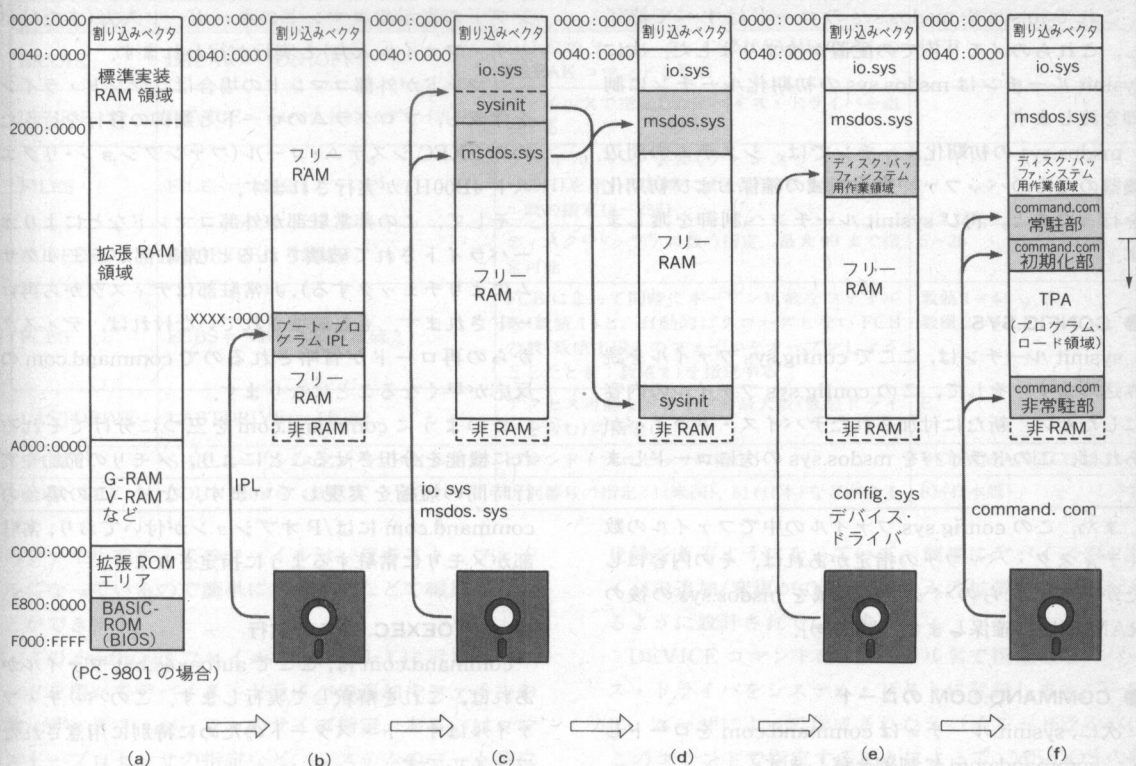
また，コマンドの解釈や実行を行うcommand.comをmsdos.sysから切り放して，別のファイル(モジュール)にしておくことにより，このcommand.comの機能の変更やバージョン・アップなどの際にも，command.comのファイルを変えるだけですみ，マン・マシン・インターフェース機能の変更が容易に実現できるというメリットも生まれてきます．

ユーザは，この機能により自分でcommand.comを作成し，UNIXにおけるシェルのような高機能なコマンド・プロセッサと取り替えて使用することも可能になっています．

■ ブート手順

さて，それではこれらのファイル(機能)が，システムのブート時にどのような手順でメモリに対してロードされ，またシステムのメモリ・マップはどのようになっているのでしょうか．

図3-2は，MS-DOSのブート手順を表しています．以下，これにそって解説していきます．

〔図3-1〕MS-DOSの基本構成

〔図3-2〕MS-DOSシステムのブート手順

● メモリ環境

8086 CPUでは256個の割り込みをサポートしています．そして，それぞれセグメント・アドレスとオフセット・アドレスをもっていることから，この割り込みベクタのテーブルとして0400H(1Kバイト)のメモリを占有します．この1Kバイトのテーブルは，セグメント・アドレス0000Hから配置されるので，MS-DOSのシステムはセグメント・アドレス0040Hから配置されます．

したがって，MS-DOSのメモリの実行環境としては，セグメント・アドレス0000Hから連続してRAMが存在しなければなりません．また通常，メモリの最上位にはブート・プログラムやBIOSの入っているROMが存在することになります．

● IPL

図3-2(a)はPC-9801シリーズにおけるメモリ・マップです．同図のROMの中には，電源ON時にシステムの状態をチェックするプログラムやBIOSが入っています．このシステム診断プログラムにより，まず周辺機器のチェックやメモリのチェックが行われたあと，ROM-BIOSを使って，MS-DOSのシステム・ディスクからイニシャル・プログラム・ローダ(Initial Program Loder：IPL)がロードされます．そして，ROM内のプログラムからこのIPLにJMPして，制御がこのIPLに移ります［同図(b)］．

● IO.SYS/MSDOS.SYSのロード

次に，このIPLによりMS-DOSのシステム・ディスクからio.sysとmsdos.sysがメモリ中にロードされます［同図(c)］．

このうちio.sysには，大きく分けてシステム常駐部と，システムの初期化のみに使われる初期化ルーチン(sysinit)の二つのルーチンが入っています．そして，このio.sysの常駐部にプログラムの制御が移ります．

io.sysの常駐部では，接続されている周辺機器の有無や種類，あるいはRAMの実装状態などをチェックします．これらの機器のチェックやRAMの状態を把握することによって，各ドライバ・ルーチンの初期化が可能になります．

このあと，制御はio.sys内のsysinitルーチンに移ります．sysinitルーチンでは，自分自身をRAMの最上位に移動し，そのRAMの最上位に移動されたsysinitルーチンで，msdos.sysをio.sysの常駐部の後に移動します．［同図(d)］．

● 初期化

これでio.sysやmsdos.sysのロードはすべて終了し，これらのメモリ内での配置が決まりました．次に，sysinitルーチンはmsdos.sysの初期化ルーチンに制御を渡します．

msdos.sysの初期化ルーチンでは，システムの周辺機器のためのバッファや作業領域の確保および初期化を行ったあと，再びsysinitルーチンへ制御を渡します．

● CONFIG.SYS

sysinitルーチンは，ここでconfig.sysファイルを読み込みます．そして，このconfig.sysファイルの内容にしたがい，新たに付加されたデバイス・ドライバがあれば，このドライバをmsdos.sysの次にロードします．

また，このconfig.sysファイルの中でファイルの数やディスク・バッファの指定があれば，その内容にしたがってこれらのバッファ領域をmsdos.sysの後のRAM領域に確保します［同図(e)］．

● COMMAND.COMのロード

次に，sysinitルーチンはcommand.comをロードして，command.comに制御を移します．

command.comは常駐部と初期化部および非常駐部の三つの部分から構成されています．その理由は，ver. 3.30ではcommand.comが全体で24Kバイト以上もの大きさであり，これらがすべてメモリに常駐してしまうと，RAMのユーザ領域(TPA：Transient Program Area)が少なくなり，ユーザの使用できるメイン・メモリが少なくなってしまいます．

そこで，command.comを常駐部と初期化部および非常駐部の三つに分けてメモリ内に配置します［同図(f)］．常駐部は，INT22H～INT24Hハンドラや非常駐部のブート・ルーチンなどから構成されています．また，MS-DOSの標準エラー・ハンドリングもこのcommand.comの常駐部によって処理されます．

次に，command.comの初期化部は常駐部の後にロードされ，システム起動の時点のみ利用されます．この部分にはautoexec.batの処理ルーチンが入っていて，プログラムのロードが可能なセグメント・アドレス(TPA：プログラム・セグメントとも呼ぶ)はこの初期化部によって決定されます．また，この初期化部はそのあとの処理では必要ないので，最初にロードされる外部コマンドによってオーバライトされて破壊されます．

非常駐部は，メモリの最上位にロードされ，この部分にはすべての内部コマンドとバッチ・ファイル・プロセッサが入っています．この非常駐部により，プロンプトの表示やコマンドのキーボード入力(またはバッチ・ファイル入力)と実行が行われます．

コマンドが外部コマンドの場合はコマンド・ラインを作成し，プログラムのロードと制御の移行を行うためにEXECシステム・コール(ファンクション・リクエスト4B00H)が実行されます．

そして，この非常駐部が外部コマンドなどによりオーバライトされて破壊されると(常駐部がチェックサムによりチェックする)，非常駐部はディスクから再ロードされます．もし破壊されていなければ，ディスクからの再ロードが省略されるのでcommand.comの反応が早くなることになります．

このようにcommand.comを三つに分けてそれぞれに機能を分担させることにより，メモリの節約や実行時間の短縮を実現しています．なお，この場合のcommand.comには/Pオプションが付いており，常駐部がメモリに常駐するように指定されます．

● AUTOEXEC.BATの実行

command.comは，ここでautoexec.batファイルがあれば，これを解釈して実行します．このバッチ・ファイルはオート・スタートのために特別に用意されたファイルです．

ユーザが電源ON時に自動的にプログラムを実行したい場合には，このバッチ・ファイルにその必要なコマンド操作の手続きを書いておけば，システム起動時に自動的に実行されることになります．

また，このファイルはシステム設定(speedコマンドやpromptコマンド，pathコマンド，cdコマンドなど)を行う際にも上手に活用したいファイルです．

このバッチ・ファイルの解釈実行が終了して，command.comはユーザの入力待ちになります．

■ config.sysファイル

MS-DOSでは，起動時に使用するシステム構成(Configuration)を指定するためのシステム構築ファイルとして“config.sys”と呼ばれるファイルを用意しています．

たとえば，使用するプリンタ(LBPなど)に応じたデバイス・ドライバや，マウスを利用するためのマウス・ドライバを用意する場合には，このconfig.sysファイルにコマンドを使って必要なドライバを登録することが可能になります．

このconfig.sysファイルは，前述のようにMS-DOSのブート時にsysinitルーチンによって参照され，MS-DOS内のシステムの細かい設定を指定するため

〔表3-1〕 CONFIG.SYS内で利用できるコマンド

コマンド	書式	機能	デフォルト
BREAK	BREAK= [ON|OFF]	ctrl-C のチェック．COMMAND.COM の BREAK コマンドと等価	OFF
DEVICE	DEVICE= [〈パス名〉]〈ファイル名〉	ファイル名で指定したデバイス・ドライバを追加する	
FILES	FILES=〈数値〉	ファイル・ハンドル(ファンクション 2FH ~60H)を用いて同時にオープンできるファイル数の指定(8~255)	8
BUFFERS	BUFFERS=〈数値〉	ディスク・バッファの数の指定．最大 99 まで指定可能	5~20
FCBS	FCBS=〈数値1〉,〈数値2〉	FCB によって同時にオープン可能なファイル数(数値1)と，自動的にクローズしない FCB の数(数値1以上のファイルをオープンしようとしたとき：数値2)を指定する	数値1=4 数値2=0
LASTDRIVE	LASTDRIVE=〈英字〉	アクセス可能なドライブの最大数(仮想ドライブを含む)の指定(A~Z)	E
SHELL	SHELL= [〈パス名〉]〈ファイル名〉	コマンド・プロセッサの指定	¥COMMAND.COM/P
COUNTRY	COUNTRY=〈数値〉	国別番号の指定．1(米国)，81(日本)などがある	81(日本版)

のファイルです．このファイルは，テキスト・ファイルになっているので簡単にエディタなどで編集することができます．

このconfig.sysファイルでは，**表3-1**に示したコマンドを用いてデバイス・ドライバの追加やファイルの数，ディスク・バッファのサイズ指定，あるいはコマンド・プロセッサの指定など，システムのブート時に必要な(可変できる)内容を指定することができます．

同表のコマンドのうち，LASTDRIVEコマンドとFCBSコマンドはver.3.10になってから追加されたものです．

● BREAKコマンド

MS-DOSでは，プログラムの実行を中止する場合には^C(コントロールC：CTRLキーとCを同時に押す)を入力します．この^C入力は，BREAKフラグの状態によって受け付けられる場合が区別されています．

通常，BREAKフラグはOFFに設定されていて，^C入力はコンソール入出力時かプリンタ出力時にのみ受け付けられます．これに対し，このBREAKコマンドでONを指定すると，ディスクの入出力を含めたすべてのシステム・コールにおいて^C入力が受け付けられるようになります．

● DEVICEコマンド

新しい入出力デバイスを追加する場合に，それらのドライバ・ルーチンを，ユーザがそのつどio.sysに組み込むようにすると，ユーザ側でio.sysの改造や組み込みの手間暇がかかり好ましくありません．

MS-DOSでは，このDEVICEコマンドを用いてconfig.sysファイルにユーザのデバイス・ドライバを登録できるようになっていて，簡単にデバイスやドライバの追加/変更ができ，システムの拡張が容易に行えるように設計されています．

DEVICEコマンドは，ファイル名で指定したデバイス・ドライバをシステム・リストに登録します．一般に，ユーザによって作成されたデバイス・ドライバは，このコマンドで指定することによって，MS-DOSの起動時にそのデバイス・ドライバをシステムに追加して利用することが可能となります．

ver.3.10以上では，プリンタ・ドライバやRS-232Cドライバはio.sysに組み込まれていないため，このDEVICEコマンドによって必要なデバイス・ドライバの登録を行って，システムの環境を整えなければなりません．

● FILESコマンド

このコマンドは，ファイル・ハンドルを用いたシステム・コールにおいて，一度にオープンできるファイルの数を指定します．このファイルの数はデフォルトで8になっていて，ユーザは何も指定しなくても8個までのファイルはオープン可能です．

ここで，**表3-2**に示した標準入出力(5個のファイル・ハンドル)は，システムを起動した時点において自動的にオープンされ，いつでも使用可能となっているため，ユーザが新しくオープンできるファイルの数はデフォルトでは3個となっています．

このファイルの数は最大99個まで指定できますが，実際には1プロセス(プログラム)当たり最大20個のファイルまでしか扱うことができないので，これ以上の数を設定しても無意味です．

〔表3-2〕標準入出力とファイル・ハンドル

標準入出力	番号	機　　能
標準入力	0	通常はコンソールになっているが，I/Oリダイレクト機能によりファイルにすることができる．プログラムへの入力を決める
標準出力	1	通常はコンソールになっている．プログラムからの出力を決める．I/Oリダイレクト機能によりファイルにすることも可能
標準エラー出力	2	常にコンソールになっている．MASMやLINKなどのエラー・メッセージの出力はこれを使用している
補助入出力	3	RS-232Cを指す．デバイス名はAUX．機械語によりI/Oリダイレクトも可能になる
標準プリンタ	4	デバイス名はPRN．機械語によりI/Oリダイレクトも可能である

● BUFFERSコマンド

このコマンドはディスク・バッファのサイズを指定します．ディスク・バッファとは，ディスクに対して読み書きするデータを一時的に保存しておくためのメモリ領域をいいます．

このBUFFERSコマンドで指定する数値は，バッファのバイト数ではなくバッファの数であり，そのバイト数はシステムのディスク構成によって異なってきます．1バッファ当たりのバイト数は，640Kバイトのディスクを装備したシステムでは512バイトであり，1Mバイトのディスクを装備したシステムでは1024バイトとなります．バッファの数のデフォルト値は，**表3-3**に示したようにシステムのメモリ実装状況によって自動的に決定されます．

このバッファの数は最大99まで指定できますが，あまり多くするとメモリ中のバッファ領域が大きくなり，その分だけTPA(プログラム・エリア)が狭くなるため，むやみに大きくは取れないことになります．このバッファ容量の指定を行う際には，この点を考慮してそのシステムに適した数を指定すべきでしょう．

筆者の経験では，PC-9801上でコンパイラを用いてプログラム開発を行った際に，ファイル数やバッファ容量の値を適当に設定したところ，デフォルト値で使用するよりも5倍以上の処理速度が改善された例があります．したがって，これらの値をシステムに合わせて適切に設定することは，処理速度のうえからは非常に重要な要素になります．

〔表3-3〕メモリ容量とバッファの数

メモリ容量(Kバイト)	デフォルト
384	5
512	10
640	20

しかし，ディスク・バッファをサポートしていないシステム(io.sys)では，このバッファを増やしても無意味なことが多いので注意すべきです(後述)．

● FCBSコマンド

FCBSコマンドはver.3.10になってから追加されたコマンドで，FCB(File Control Block)を用いてオープンできるFCBの最大数(1～255)，および自動的にクローズされないFCBの数を指定します．

ver.2.11では，FCBによって無制限にファイルをオープンすることが可能となっていました．しかし，ver.3.10以降では，MS-Networksの環境下においてファイルのオープン/クローズをすべてOS側で把握しておく必要があるため，FCBによるファイル・オープンでもその管理領域が必要となってきます．FCBSコマンドは，このFCBのための管理領域の数を指定し，FCBを用いて同時にオープン可能なファイルの数を定義します．

また，MS-DOSではFCBを用いて同時にオープン可能なファイルの数を越えたファイルをオープンしようとした際に，最後にアクセスされた(時間が最も古い)ファイルから順番に自動的にクローズしていきます．このとき，FCBSコマンドの第2パラメータ"自動的にクローズされないFCB数"で指定された数のファイルはクローズされません．

● LASTDRIVEコマンド

ver.3.10以降において，MS-Networksの環境下では，ネットワーク回線を通じて接続されているドライブ(リモート・ドライブ)を仮想ドライブとして割り当てることが可能となりました(substコマンドなど)．LASTDRIVEコマンドは，この際に使用する仮想ドライブの最終値を設定するのに使用されます．

この最終値のデフォルトはEドライブになっているため，もしLASTDRIVEの指定がない場合でも五つのドライブ(A～E)までは使用可能となります．また，五つ以上のドライブが接続されている場合でも，接続されているすべてのドライブは自動的に認識されます．

● SHELLコマンド

このコマンドは，MS-DOSシステムで使用するコマンド・プロセッサを指定するためのコマンドです．デフォルトではcommand.comがブート・ディスクのルートディレクトリからロードされます．

SHELLコマンドで与えられたコマンド・プロセッ

〔リスト3-1〕SHELLコマンドによる再ロード・パスの指定

```
R>type a:¥config.sys ⏎ … config.sysファイルの内容を確認する①
DEVICE=PRINT.SYS
DEVICE=RSDRV.SYS                 (最初のブート)
FILES=30                                      (メモリに常駐)
BUFFERS=30
SHELL=C:¥COM3¥COMMAND.COM A: /P                 ②
LASTDRIVE=Z
                                  (非常駐部のブート)
R>set ⏎ … 環境変数の確認③
COMSPEC=A:¥COMMAND.COM ←④
PATH=R:¥;H:¥BIN;H:¥MSC¥EXE;H:¥COM3;H:¥BAT
HELP=H:¥BIN
PROFILE=H:¥BIN
INCLUDE=H:¥MSC¥INCLUDE¥
MIMACRO=H:¥BIN

R>symdeb ⏎ … デバッガの起動⑤
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-d 9f00:0 ⏎ … command.comの非常駐部の一部を確認⑥
9F00:0000  0D 0A 00 0A 3C 3C 6F 66-66 3E 00 15 3C 3C 6F 6E   ....<<off>..<<on
9F00:0010  3E 00 1D 3C 83 70 83 58-82 CC 8E 77 92 E8 82 AA   >..<.p.X.L.w.h.*
                              .
                              .
-f 9f00:0 1300 0 ⏎ … 非常駐部を破壊⑦
-d 9f00:0 ⏎ … オーバライトの確認⑧
9F00:0000  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
9F00:0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
                              .
                              .
} … ここでドライブAのドアを開ける⑨
-q ⏎ … symdebの終了⑩

COMMAND.COMをロードできません.

ドライブの準備ができていません.<読取り中><ドライブ A:>      } command.comの非常駐部が破壊されているので
¥COMMAND.COMの入ったディスクをカレントドライブに差し込み,   } リブートしようとしたが失敗⑪
どれかキーを押してください.
} … ここでドライブAのドアを閉じて適当なキーを押す⑫
R> … command.comが正常に再ロードされた⑬
```

サのファイル名(パス名を含む)は,後述する環境変数のCOMSPECにセットされます.command.comの非常駐部がオーバライトされた場合は,この環境変数COMSPECにセットされたファイル名によってcommand.comのリブートが行われます.

【SHELLコマンドの実行サンプル】

リスト3-1は,config.sysファイルにおけるSHELLコマンドの使用例とcommand.comの再ロードの例を示しています.

① まず,config.sysファイルの内容を確認する.

② ここでは,command.comに対してリブートの際のパス(ドライブ名を含む)や/Pオプションを指定している.command.comの/Pオプションは,常駐部をメモリに常駐させるためのオプション.

③ 次に,SETコマンドを用いて環境変数を調べる.

④ すると,環境変数COMSPECには,SHELLコマンドで指定したcommand.comのリブート・パス(A:¥)が設定されているのが確認できる.

⑤ メモリの操作を行うためにデバッガsymdebを起動する.

⑥ 次に,dコマンドを用いてcommand.comの非常駐部の一部を確認する.

⑦ 次に,fコマンドを用いてcommand.comの非常駐部をオーバライトして破壊する.

⑧ dコマンドを用いてオーバライトされたことを確認する.

⑨ ここで,デバッガsymdebを終わるまえにドライブAのドアを開けておく.

⑩ qコマンドでデバッガsymdebを終了する.

⑪ ここで,command.comの非常駐部が破壊されているため,環境変数COMSPECにしたがってドライブAから再ロードしようと試みるが,ドライブAのドアが開いているためエラーが発生し,その旨のメッセージが表示される.

⑫ そこで,ドライブAのドアを閉めて適当なキーを押す.

⑬ すると,command.comの非常駐部が正常に再ロードされる.

〔リスト3-2〕
COUNTRY コマンドによる表示の相違

```
R>type a:¥config.sys ⏎ …comfig.sys ファイルの内容を確認
DEVICE=PRINT.SYS
DEVICE=RSDRV.SYS
FILES=30
BUFFERS=30
SHELL=C:¥COM3¥COMMAND.COM /P
LASTDRIVE=Z
COUNTRY=1 ←── (米国フォーマットの指定)

R>date ⏎
現在の日付は (月) 2-06-1989 です.        ⎫ 米国フォーマット
日付を入力してください (mm-dd-yy):       ⎭

R>dir h:¥wk1¥masm ⏎ …dirコマンドの実行

 ドライブ H: のディスクのボリュームラベルは HD1
 ディレクトリは H:¥WK1¥MASM

.              <DIR>      1-31-89    8:30a
..             <DIR>      1-31-89    8:30a
SOURCE   ASM       806    1-31-89    3:05p
TOKEN    ASM       692    1-31-89    3:06p
TOKEN    LST      2026    1-31-89    9:39a
SOURCE   LST      1259    1-31-89    3:06p ←── (米国フォーマット)
                  .
                  .
                  .
OPE      ASM      1437    2-03-89    5:05p
OPE      LST      3038    2-03-89    4:39p
OPE      OUT      1989    2-03-89    3:51p
        98 個のファイルがあります.
    262144 バイトが使用可能です

R>
```

● **COUNTRY コマンド**

MS-DOSが保存している国別情報を指定するためのコマンドとしてCOUNTRYがあります．国番号としては，1(アメリカ)か81(日本)が用意されています．これによりdirコマンドから出力される日付のフォーマットが，それぞれの国のものに変わります(**リスト3-2**)．

3-2 プログラムのロードと実行

さて，システムのロードが終わりコマンド待ちの状態から，今度はコマンド操作によって入力されたコマンドが外部コマンドであれば，そのコマンドがメモリ中にロードされ実行されることになります．ここでは，これらMS-DOSの外部コマンド(すなわちプログラム)の実行のシーケンスを追っていきます．

■ プロセスの起動

MS-DOSでは，メモリをパラグラフ単位(上位16ビット)で管理しており，**図3-3**のようにメモリ管理情報，環境，PSP，プログラムやデータを一つのブロックとして扱い，これらのブロックをメモリ管理情報内のポインタでリンクすることにより管理しています．

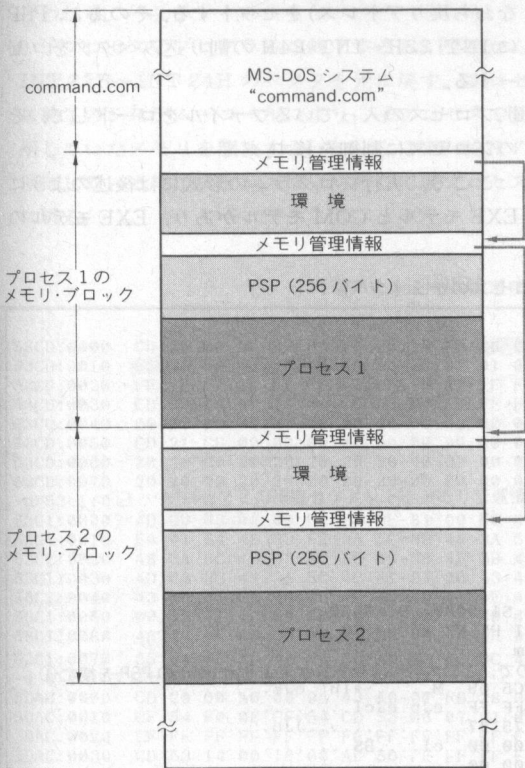
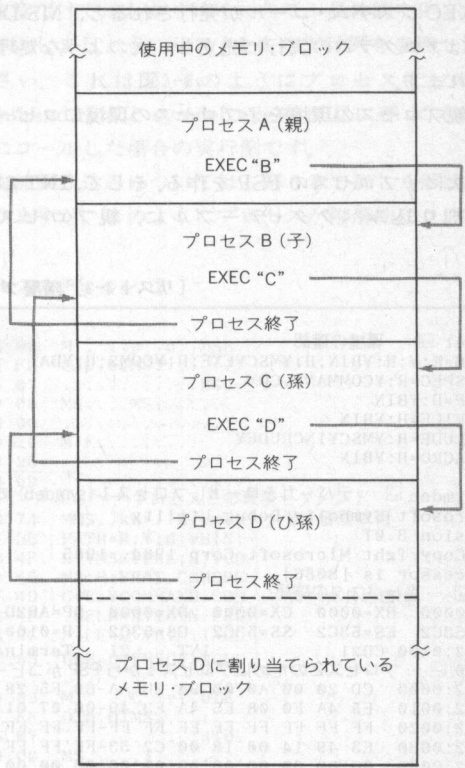
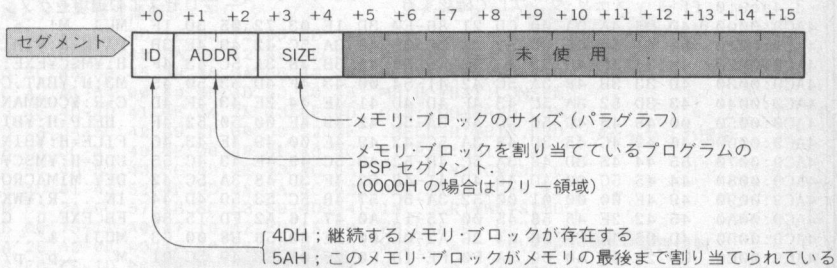
環境とは，UNIXから受け継いだ概念で，SETコマンドなどで定義した文字列(環境変数)の集まりをいいます．

また，PSP(Program Segment Prefix)とは，MS-DOSとユーザ・プログラムとのインターフェースを取るための領域であり，このPSPや環境についてはこの後で詳しく述べてあります(95，97ページ)．

メモリ管理情報は，**図3-4**に示したように16バイトで構成されます．ここには，メモリ・ブロックの先頭アドレスやサイズなど，MS-DOSがメモリを管理するための情報が入っています．

MS-DOSでは，プログラムのことを“プロセス”と呼んでいますが，このプロセスを次々に呼んでチェインすることができるようになっています．つまりあるプロセスの中で別のプロセス(子プロセス)を呼び，さらにその子プロセスの中で孫プロセスを呼ぶというように，プログラムを階層的に呼ぶことができるのです(**図3-5**)．

MS-DOSから，コマンド・プロセッサ(command.com)を呼ぶ際にも，同様の手法により子プロセスとしてコマンド・プロセッサのロードおよび実行が行われます．また，外部コマンドを呼ぶ際にも同様の手法を用いて，command.comの子プロセスとしてロードおよび実行が行われています．

〔図3-3〕プロセス管理の単位

〔図3-5〕プロセスの階層的な起動

〔図3-4〕メモリ管理情報

この子プロセスの起動を行うサービス・ルーチンはEXECシステム・コールと呼ばれ，MS-DOSの中に含まれています．EXECシステム・コールについては，第6章でさらに詳しく解説してありますが，ここでも簡単にこのEXECシステム・コールの約束について触れておくことにします．

EXECシステム・コール

MS-DOSの場合，ロードされるプロセス(プログラム)が使用できるメモリ・エリアとしてRAMの最上限までが割り当てられます．したがって，EXECシステム・コールで子プロセスを起動するまえに，いくらかのメモリを開放しなければなりません．

そこでcommand.comでは，ファンクション・リクエスト4AHを用いて，自分自身の直後のメモリからメモリの最上限までを開放してMS-DOSの管理下に戻します．そして，このcommand.comの直後の開放されたメモリ領域に子プロセスをロードして起動します．

ここで，プロセスをロードすべきメモリの空き領域はプログラム・セグメントと呼ばれています．このプログラム・セグメントの先頭256バイトはPSPと呼ばれます．PSPはMS-DOSとプロセスの橋渡しをする領域であり，親プロセス(command.com)から渡されたパラメータ文字列や各種の情報が格納されています

(PSPの構成：95ページ).

EXECシステム・コールが発行されると，MS-DOS内ではプログラムの実行に先立ち，次のような処理が行われます.

(1) 親プロセスの環境を子プロセスの環境にコピーする.

(2) 次に子プロセスのPSPを作る. そして，INT 22Hの割り込みベクタ・テーブルに，親プロセスのEXECシステム・コールの次の命令のアドレス(すなわち戻りアドレス)をセットする. そのあと，PSPにINT 22H～INT 24Hの割り込みベクタをコピーする.

(3) プロセスの入っているファイルをロードして，そのプロセスに制御を移す.

ここで，実行プログラムの形式には後述のようにEXEモデルとCOMモデルがあり，EXEモデルの

〔リスト3-3〕階層プロセスとプロセスのチェイン ①

```
R>set ⏎ … 環境の確認
PATH=R:¥;H:¥BIN;H:¥MSC¥EXE;H:¥COM3;H:¥BAT
COMSPEC=R:¥COMMAND.COM
HELP=H:¥BIN
PROFILE=H:¥BIN
INCLUDE=H:¥MSC¥INCLUDE¥
MIMACRO=H:¥BIN

R>symdeb ⏎ … デバッガをロードしプロセス1(symdeb)のPSPを調べる
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-r ⏎ … 各レジスタの確認
AX=0000  BX=0000  CX=0000  DX=0000  SP=AB2D  BP=0000  SI=0000  DI=0000
DS=53C2  ES=53C2  SS=53C2  CS=53C2  IP=0100   NV UP EI PL NZ NA PO NC
53C2:0100 CD21          INT    21  ;Terminate Program
-d 0 ⏎ … プロセス2のためにプロセス1からPSPがコピーされているので，この内容によりプロセス1(symdeb)のPSPを捜す ①
53C2:0000   CD 20 00 A0 00 9A 2D AA-69 F5 28 09 E5 4A C5 09   M . ..-*iu(.eJE.
53C2:0010   E5 4A F0 08 E5 4A E3 49-06 07 01 00 02 FF FF FF   eJp.eJcI........
53C2:0020   FF FF FF FF FF FF FF FF-FF FF FF FF CA 4A 75 07   ............JJu.
53C2:0030   E3 49 14 00 18 00 C2 53-FF FF FF FF 01 00 00 00   cI....BS........
53C2:0040   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
53C2:0050   CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20   M!K.........
53C2:0060   20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20          .....
53C2:0070   20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00          ........
                                                   └ プロセス1の環境セグメント ②
-d 4ac9:0 ff ⏎ … メモリ・ダンプして確認する
4AC9:0000   4D D5 4A 0A 00 CD 21 86-E0 3D 1E 03 72 05 3D 1F   MUJ..M!.`=..r.=.} メモリ管理情報
4AC9:0010   50 41 54 48 3D 52 3A 5C-3B 48 3A 5C 42 49 4E 3B   PATH=R:¥;H:¥BIN;
4AC9:0020   48 3A 5C 4D 53 43 5C 45-58 45 3B 48 3A 5C 43 4F   H:¥MSC¥EXE;H:¥CO
4AC9:0030   4D 33 3B 48 3A 5C 42 41-54 00 43 4F 4D 53 50 45   M3;H:¥BAT.COMSPE
4AC9:0040   43 3D 52 3A 5C 43 4F 4D-4D 41 4E 44 2E 43 4F 4D   C=R:¥COMMAND.COM
4AC9:0050   00 48 45 4C 50 3D 48 3A-5C 42 49 4E 00 50 52 4F   .HELP=H:¥BIN.PRO  } プロセスの環境が
4AC9:0060   46 49 4C 45 3D 48 3A 5C-42 49 4E 00 49 4E 43 4C   FILE=H:¥BIN.INCL  } コピーされている ④
4AC9:0070   55 44 45 3D 48 3A 5C 4D-53 43 5C 49 4E 43 4C 55   UDE=H:¥MSC¥INCLU
4AC9:0080   44 45 5C 00 4D 49 4D 41-43 52 4F 3D 48 3A 5C 42   DE¥.MIMACRO=H:¥B
4AC9:0090   49 4E 00 00 01 00 52 3A-5C 57 4B 5C 53 59 4D 44   IN....R:¥WK¥SYMD
4AC9:00A0   45 42 2E 45 58 45 00 75-11 A0 47 16 A2 FD 15 80   EB.EXE.u. G."}..
4AC9:00B0   4D D5 4A EC 08 04 26 A2-0E 00 1E 50 56 B8 00 63   MUJl..&"...PV8.c} メモリ管理情報 ③
4AC9:00C0   CD 20 00 A0 00 9A F0 FE-1D F0 2F 01 E3 49 3C 01   M . ..p~.p/.cI<.
4AC9:00D0   E3 49 EB 04 E3 49 E3 49-06 07 01 00 02 FF FF FF   cIk.cIcI........  } プロセス1のPSP
4AC9:00E0   FF FF FF FF FF FF FF FF-FF FF FF FF CA 4A 8C 8D   ............JJ..
4AC9:00F0   E5 4A 14 00 18 00 D5 4A-FF FF FF FF 01 00 00 00   eJ....UJ........
-q ⏎        └ハード・エラー       └プログラム      └ctrl-Cによる
             処理アドレス(INT 24H)   終了アドレス     中断処理アドレス(INT 23H)
R>symdeb command.com ⏎ … プロセス2(command. com)  (INT 22H)
Microsoft Symbolic Debug Utility        のロード
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-r ⏎ … プロセス2の開始セグメントの確認
AX=0000  BX=0000  CX=6163  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=53CD  ES=53CD  SS=53CD  CS=53CD  IP=0100   NV UP EI PL NZ NA PO NC
53CD:0100 E95D0D        JMP    0E60
-g ⏎ … プロセス2の実行

Command バージョン 3.30

R>symdeb ⏎ … プロセス3(symdeb)によりプロセス2とプロセス3のPSPを調べる
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-d 53cd:0 ⏎ … 確認しておいたセグメント・アドレスからプロセス2のPSPを調べる ⑤
```

場合には，実行のまえにプログラムのリロケートが行われる．

(4) 子プロセスでプログラムが終了すると，PSPからINT 22H〜INT 24Hのベクタを元に戻す．プロセス終了のシステム・コールにより，子プロセスで使用していたメモリを開放する．そして，INT 22Hの保持している終了アドレスにJMPして親プロセスに戻る．

【EXECシステム・コールの実行サンプル】

文章だけではわかりにくいので，具体的な例を示して解説しましょう．**リスト3-3**の実行例を参照してください．これは**図3-6**のようにプロセス(ここではcommand.comとsymdeb.exeを交互に呼ぶ)を階層的にコールした場合の実行例です．

内部割り込みについては，第5章で詳しく解説してありますが，このようにプロセスが階層的にコールさ

〔リスト3-3〕階層プロセスとプロセスのチェイン ②

```
53CD:0000   CD 20 00 A0 00 9A F0 FE-1D F0 88 02 CD 53 C5 09   M . ..p~.p..MSE.
53CD:0010   E5 4A F0 08 E5 4A CD 53-06 07 01 00 02 FF FF FF   eJp.eJMS........
53CD:0020   FF FF FF FF FF FF FF FF-FF FF FF FF C2 53 75 07   ............BSu.
53CD:0030   CD 53 14 00 18 00 CD 53-FF FF FF FF 01 00 00 00   MS...MS.........
53CD:0040   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
53CD:0050   CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20   M!K..........
53CD:0060   20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20          .....
53CD:0070   20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00           ........
-d 53c1:0 ⏎ … プロセス2の環境セグメントとメモリ管理情報のメモリ・ダンプ⑥   プロセス2の環境セグメント
53C1:0000 4D CD 53 0A 00 EB C8 BE-81 00 E8 09 F0 3C 0D 74   MMS..kH>..h.p<.t } メモリ管理情報
53C1:0010   50 41 54 48 3D 52 3A 5C-3B 48 3A 5C 42 49 4E 3B   PATH=R:¥;H:¥BIN;  )
53C1:0020   48 3A 5C 4D 53 43 5C 45-58 45 3B 48 3A 5C 43 4F   H:¥MSC¥EXE;H:¥CO  |
53C1:0030   4D 33 3B 48 3A 5C 42 41-54 00 43 4F 4D 53 50 45   M3;H:¥BAT.COMSPE  |
53C1:0040   43 3D 52 3A 5C 43 4F 4D-4D 41 4E 44 2E 43 4F 4D   C=R:¥COMMAND.COM  } プロセス2の環境
53C1:0050   00 48 45 4C 50 3D 48 3A-5C 42 49 4E 00 50 52 4F   .HELP=H:¥BIN.PRO  |
53C1:0060   46 49 4C 45 3D 48 3A 5C-42 49 4E 00 49 4E 43 4C   FILE=H:¥BIN.INCL  |
53C1:0070   55 44 45 3D 48 3A 5C 4D-53 43 5C 49 4E 43 4C 55   UDE=H:¥MSC¥INCLU  )
-d 0 ⏎ … プロセス3のPSPを調べる
5DAC:0000   CD 20 00 A0 00 9A 43 A0-07 F6 28 09 CF 54 C5 09   M . ..C .v(.OTE.
5DAC:0010   CF 54 F0 08 CF 54 CD 53-06 07 01 00 02 FF FF FF   OTp.OTMS........
5DAC:0020   FF FF FF FF FF FF FF FF-FF FF FF FF B4 54 75 07   ............4Tu.
5DAC:0030   CD 53 14 00 18 00 AC 5D-FF FF FF FF 01 00 00 00   MS....,]........
5DAC:0040   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
5DAC:0050   CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20   M!K..........
5DAC:0060   20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20          .....
5DAC:0070   20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00           ........
-d 54b3:0 ff ⏎ … プロセス3の環境とメモリ管理情報のメモリ・ダンプ⑦   プロセス3の環境セグメント
54B3:0000 4D BF 54 0A 00 CD 21 86-E0 3D 1E 03 72 05 3D 1F   M?T..M!. =..r.=. } メモリ管理情報
54B3:0010   50 41 54 48 3D 52 3A 5C-3B 48 3A 5C 42 49 4E 3B   PATH=R:¥;H:¥BIN;  )
54B3:0020   48 3A 5C 4D 53 43 5C 45-58 45 3B 48 3A 5C 43 4F   H:¥MSC¥EXE;H:¥CO  |
54B3:0030   4D 33 3B 48 3A 5C 42 41-54 00 43 4F 4D 53 50 45   M3;H:¥BAT.COMSPE  |
54B3:0040   43 3D 52 3A 5C 43 4F 4D-4D 41 4E 44 2E 43 4F 4D   C=R:¥COMMAND.COM  |
54B3:0050   00 48 45 4C 50 3D 48 3A-5C 42 49 4E 00 50 52 4F   .HELP=H:¥BIN.PRO  |
54B3:0060   46 49 4C 45 3D 48 3A 5C-42 49 4E 00 49 4E 43 4C   FILE=H:¥BIN.INCL  } プロセス3の環境
54B3:0070   55 44 45 3D 48 3A 5C 4D-53 43 5C 49 4E 43 4C 55   UDE=H:¥MSC¥INCLU  |
54B3:0080   44 45 5C 00 4D 49 4D 41-43 52 4F 3D 48 3A 5C 42   DE¥.MIMACRO=H:¥B  |
54B3:0090   49 4E 00 00 01 00 52 3A-5C 57 4B 5C 53 59 4D 44   IN....R:¥WK¥SYMD  |
54B3:00A0   45 42 2E 45 58 45 00 75-11 A0 47 16 A2 FD 15 80   EB.EXE.u. G."}..  )
54B3:00B0 4D BF 54 EC 08 04 26 A2-0E 00 1E 50 56 B8 00 63   M?T1..&"...PV8.c } メモリ管理情報
54B3:00C0   CD 20 00 A0 00 9A F0 FE-1D F0 2F 01 CD 53 3C 01   M . ..p~.p/.MS<.  )
54B3:00D0   CD 53 EB 04 CD 53 CD 53-06 07 01 00 02 FF FF FF   MSk.MSMS........  } プロセス3のPSP
54B3:00E0   FF FF FF FF FF FF FF FF-FF FF FF FF B4 54 8C 8D   ............4T..  |
54B3:00F0   CF 54 14 00 18 00 BF 54-FF FF FF FF 01 00 00 00   OT....?T........  )
-q ⏎                     └INT 24H              └INT 22H └INT 23H

R>symdeb command.com ⏎ … プロセス3(symdeb)の子プロセスとしてプロセス4(command.com)
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-r ⏎ … プロセス4のセグメント・アドレスを確認
AX=0000  BX=0000  CX=6163  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=5DB7  ES=5DB7  SS=5DB7  CS=5DB7  IP=0100   NV UP EI PL NZ NA PO NC
5DB7:0100 E95D0D          JMP     0E60
-g ⏎ … プロセス4の実行

Command バージョン 3.30

R>symdeb ⏎ … デバッガによりメモリ内容を確認する
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
```

れた場合，それぞれのプロセスでは独自にINT 23H (ctrl-Cによる中断)およびINT 24H(ハード・エラー)の各処理のルーチンをもつことができます．

そして，これらの情報はプロセス起動時の割り込みベクタ・テーブルからPSPにコピーされているので，子プロセスのPSPのダンプ・リストを見れば，その親プロセスの各ルーチンの処理アドレスを知ることができます．

また，同様にPSPのオフセット0AHには子プロセスが終了したあとの戻りアドレス(親プロセス内の)がコピーされています．したがって，リストの実行例におけるプロセスと割り込みベクタのチェイン状態は，**図3-7**のように表すことができます．**リスト3-3**を追っていきましょう．

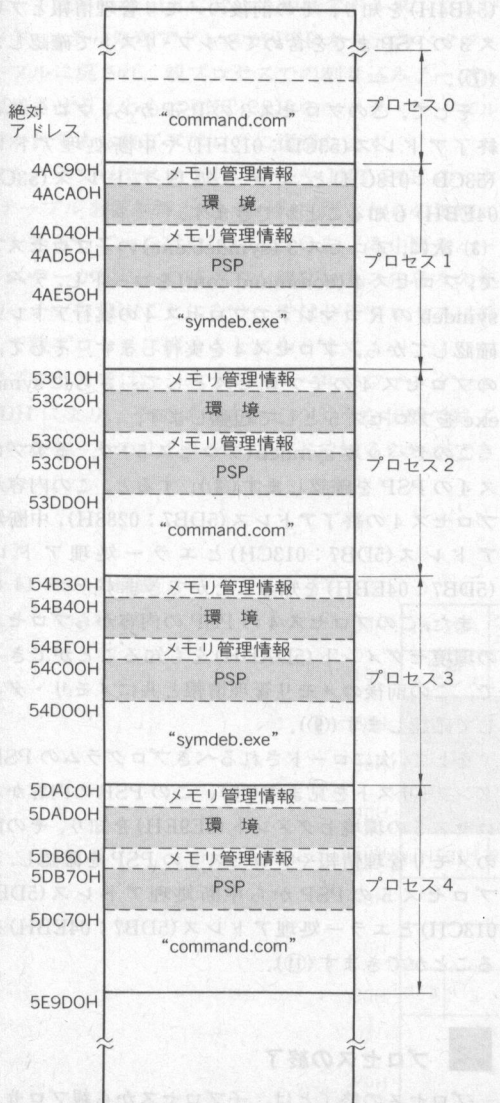
(1) システムのコマンド・プロセッサ(command.comであり，便宜上これをプロセス0とする)のsetコマン

〔リスト3-3〕階層プロセスとプロセスのチェイン ③

```
-d 5db7:0⏎ … プロセス4 (command.com)のPSPの確認⑧
5DB7:0000  CD 20 00 A0 00 9A F0 FE-1D F0 88 02 B7 5D C5 09  M . ..p^.p..7]E.
5DB7:0010  CF 54 F0 08 CF 54 B7 5D-06 07 01 00 02 FF FF FF  OTp.OT7]........
5DB7:0020  FF FF FF FF FF FF FF FF-FF FF FF FF AC 5D 75 07  ............,]u.
5DB7:0030  B7 5D 14 00 18 00 B7 5D-FF FF FF FF 01 00 00 00  7]....7]........
5DB7:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
5DB7:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20  M!K.............
5DB7:0060  20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20  ................
5DB7:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00  ................
-d 5dab:0⏎ …プロセス4の環境とメモリ管理情報のダンプ⑨   INT24H    プロセス4の環境セグメント
5DAB:0000  4D B7 5D 0A 00 00 00 00-00 00 00 00 00 00 00 00  M7].............  } メモリ管理情報
5DAB:0010  50 41 54 48 3D 52 3A 5C-3B 48 3A 5C 42 49 4E 3B  PATH=R:¥;H:¥BIN;  ┐
5DAB:0020  48 3A 5C 4D 53 43 5C 45-58 45 3B 48 3A 5C 43 4F  H:¥MSC¥EXE;H:¥CO  │
5DAB:0030  4D 33 3B 48 3A 5C 42 41-54 00 43 4F 4D 53 50 45  M3;H:¥BAT.COMSPE  │
5DAB:0040  43 3D 52 3A 5C 43 4F 4D-4D 41 4E 44 2E 43 4F 4D  C=R:¥COMMAND.COM  } プロセス4の環境
5DAB:0050  00 48 45 4C 50 3D 48 3A-5C 42 49 4E 00 50 52 4F  .HELP=H:¥BIN.PRO  │
5DAB:0060  46 49 4C 45 3D 48 3A 5C-42 49 4E 00 49 4E 43 4C  FILE=H:¥BIN.INCL  │
5DAB:0070  55 44 45 3D 48 3A 5C 4D-53 43 5C 49 4E 43 4C 55  UDE=H:¥MSC¥INCLU  ┘
-d 0⏎ … 次にロードされるべきプロセスのためのPSPを確認⑩
6796:0000  CD 20 00 A0 00 9A 59 96-A6 F6 28 09 B9 5E C5 09  M . ..Y.&v(.9^E.
6796:0010  B9 5E F0 08 B9 5E B7 5D-06 07 01 00 02 FF FF FF  9^p.9^7]........
6796:0020  FF FF FF FF FF FF FF FF-FF FF FF FF 9E 5E 75 07  .............^u.
6796:0030  B7 5D 14 00 18 00 96 67-FF FF FF FF 01 00 00 00  7]....g.........
6796:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
6796:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20  M!K.............
6796:0060  20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20  ................
6796:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00  ................
-d 5e9d:0 ff⏎ … プロセス5 (symdeb)の環境とPSPの確認⑪   環境セグメント
5E9D:0000  4D A9 5E 0A 00 CD 21 86-E0 3D 1E 03 72 05 3D 1F  M)^..M!.`=..r.=.  } メモリ管理情報
5E9D:0010  50 41 54 48 3D 52 3A 5C-3B 48 3A 5C 42 49 4E 3B  PATH=R:¥;H:¥BIN;  ┐
5E9D:0020  48 3A 5C 4D 53 43 5C 45-58 45 3B 48 3A 5C 43 4F  H:¥MSC¥EXE;H:¥CO  │
5E9D:0030  4D 33 3B 48 3A 5C 42 41-54 00 43 4F 4D 53 50 45  M3;H:¥BAT.COMSPE  │
5E9D:0040  43 3D 52 3A 5C 43 4F 4D-4D 41 4E 44 2E 43 4F 4D  C=R:¥COMMAND.COM  │
5E9D:0050  00 48 45 4C 50 3D 48 3A-5C 42 49 4E 00 50 52 4F  .HELP=H:¥BIN.PRO  │
5E9D:0060  46 49 4C 45 3D 48 3A 5C-42 49 4E 00 49 4E 43 4C  FILE=H:¥BIN.INCL  } プロセス5の環境
5E9D:0070  55 44 45 3D 48 3A 5C 4D-53 43 5C 49 4E 43 4C 55  UDE=H:¥MSC¥INCLU  │
5E9D:0080  44 45 5C 00 4D 49 4D 41-43 52 4F 3D 48 3A 5C 42  DE¥.MIMACRO=H:¥B  │
5E9D:0090  49 4E 00 00 01 00 52 3A-5C 57 4B 5C 53 59 4D 44  IN....R:¥WK¥SYMD  │
5E9D:00A0  45 42 2E 45 58 45 00 75-11 A0 47 16 A2 FD 15 80  EB.EXE.u. G."}..  ┘
5E9D:00B0  4D A9 5E EC 08 04 26 A2-0E 00 1E 50 56 B8 00 63  M)^l..&"...PV8.c} メモリ管理情報
5E9D:00C0  CD 20 00 A0 00 9A F0 FE-1D F0 2F 01 B7 5D 3C 01  M . ..p^.p/.7]<.  ┐
5E9D:00D0  B7 5D EB 04 B7 5D B7 5D-06 07 01 00 02 FF FF FF  7]k.7]7]........  } プロセス5のPSP
5E9D:00E0  FF FF FF FF FF FF FF FF-FF FF FF FF 9E 5E 8C 8D  ................  │
5E9D:00F0  B9 5E 14 00 18 00 A9 5E-FF FF FF FF 01 00 00 00  9^....)^........  ┘
-q⏎ … プロセス5からプロセス4へ   INT 24H   INT 22H   INT 23H

R>exit⏎ …プロセス4からプロセス3へ

Program terminated normally (0)
-r⏎ … アドレスの確認
AX=0000  BX=0000  CX=6163  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=5DB7  ES=5DB7  SS=5DB7  CS=5DB7  IP=0100   NV UP EI PL NZ NA PO NC
5DB7:0100 E95D0D        JMP     0E60
-q⏎ … プロセス3からプロセス2へ

R>exit⏎ … プロセス2からプロセス1へ

Program terminated normally (0)
-r⏎ … アドレスの確認
AX=0000  BX=0000  CX=6163  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=53CD  ES=53CD  SS=53CD  CS=53CD  IP=0100   NV UP EI PL NZ NA PO NC
53CD:0100 E95D0D        JMP     0E60
-q⏎ … プロセス1からプロセス0へ

R>
```

〔図3-7〕プロセスのチェインと PSP の関係▶

プロセス 0
"command.com"
49E3:012F — プロセス終了
49E3:013C — ctrl-C
49E3:04EB — ハード・エラー
4AC90H
PSP 0AH 0EH 12H
4AE50H
プロセス 1
"symdeb.exe"
4AE5:09C5 — ctrl-C
4AE5:08F0 — ハード・エラー
53C10H
PSP 0AH 0EH 12H
53DD0H
プロセス 2
"command.com"
53CD:0288 — プロセス終了
53CD:012F — プロセス終了
53CD:013C — ctrl-C
53CD:04EB — ハード・エラー
54B30H
PSP 0AH 0EH 12H
54D00H
プロセス 3
"symdeb.exe"
54CF:09C5 — ctrl-C
54CF:08F0 — ハード・エラー
5DAC0H
PSP 0AH 0EH 12H
5DC70H
プロセス 4
"command.com"
5DB7:0288 — プロセス終了
5DB7:012F — プロセス終了
5DB7:013C — ctrl-C
5DB7:04EB — ハード・エラー
5E9D0H
PSP 0AH 0EH 12H
5EB90H
プロセス 5
"symdeb.exe"

〔図3-6〕子プロセスの実行例におけるメモリ配置

ドにより，そのプロセス0の環境を確認しておきます．次に，このプロセス0の子プロセスとしてプロセス1(symdeb.exe)を起動します．ここで，このプロセス1の環境変数領域にはプロセス0の環境がコピーされることになります．

また，この時点においてデバッガ symdeb でテストされるべきプログラムのために，DS レジスタが示すアドレスのオフセット 0000H からの 256 バイトには，すでに PSP が作られていることもわかります(**リスト3-3** の①)．

次に，この PSP のオフセット 2CH を見れば symdeb.exe の環境セグメント 4ACAH(絶対アドレス 4ACA0H)を知ることができます(②)．ここでは，さらに参考のためにメモリ管理情報もメモリ・ダンプして確認しています(③)．

また環境セグメント・アドレス(4ACA0H)のメモリ内容を見ると，確かに set コマンドで確認したプロセス0(command.com)の環境がコピーされていることもわかります(④)．

絶対アドレス 4AD50H からの 256 バイトはプロセス1(symdeb.exe)の PSP です．後述のように PSP のオフセット 0AH，0EH，12H からの4バイトは，それぞれ INT 22H(プロセス終了アドレス)，INT 23H(^C による中断処理アドレス)，INT 24H(ハード・エラーによる処理アドレス)の三つのベクタ・テーブルのコピーが入ります．

これらのアドレスは，EXEC システム・コールによりプロセスが起動された時点で，ベクタ領域から PSP にコピーされ，逆にプロセスが終了したときに PSP からベクタ領域へ元の値が戻されることになります．

また，INT 23H，INT 24H の各処理はプロセスによって独自の処理ルーチンをもつことができ，その場合は対応するベクタ・テーブルを書き替えてもさしつかえありません．

したがって，プロセス1(symdeb.exe)の PSP にある終了アドレスは，このプロセスをコールした親プロセス内の戻りアドレスを保持していることになり，中断処理アドレスとエラー処理アドレスは，その親プロセスであるプロセス0が実行中のベクタ・テーブルのコピーなので，プロセス0の中断処理とエラー処理のアドレスということになります．

よって，プロセス1の PSP のオフセット 0EH にある 49E3：013CH，オフセット 12H にある 49E3：04EBH は，その親プロセスであるプロセス0(command.com)の各々の処理ルーチンのアドレスを指していることになります(**図3-7**)．

(2) 次に，このプロセス1(symdeb.exe)からプロセス2(command.com)を起動します．

そして，このプロセス2のさらに子プロセスとしてプロセス3(symdeb.exe)を実行し，このデバッガ symdeb で各々のプロセスの PSP，および環境のメモリ・ダンプを行ってその内容を確認します．

まず，プロセス2の PSP を調べると(⑤)，プロセス2の終了アドレス(53CD：0288H)，プロセス1内にある中断処理アドレス(4AE5：09C5H)，エラー処理アドレス(4AE5：08F0H)を知ることができます(**図3-7**)．

また，この PSP のオフセット 2CH の内容からプロセス2の環境セグメント(53C2H)を知り，その前後のメモリ管理情報とともに，プロセス2の環境をメモリ・ダンプして確認します(⑥)．

次に，プロセス4(次にロードすべき)の PSP から，やはりプロセス3(symdeb.exe)の環境セグメント(54B4H)を知り，その前後のメモリ管理情報とプロセス3の PSP までを含めてダンプ・リストで確認します(⑦)．

そして，このプロセス3の PSP から，プロセス3の終了アドレス(53CD：012FH)や中断処理アドレス(53CD：013CH)とエラー処理アドレス(53CD：04EBH)も知ることができます．

(3) 次に，プロセス3(symdeb.exe)の子プロセスとして，プロセス4(command.com)をロードし，デバッガ symdeb の R コマンドでプロセス4の実行アドレスを確認してから，プロセス4を実行します．そして，このプロセス4の子プロセスとして，さらに symdeb.exe をプロセス5として起動します．

このデバッガ symdeb(プロセス5)で，まずプロセス4の PSP を確認します(⑧)．すると，この内容からプロセス4の終了アドレス(5DB7：0288H)，中断処理アドレス(5DB7：013CH)とエラー処理アドレス(5DB7：04EBH)を知ることができます．

また，このプロセス4の PSP の内容からプロセス4の環境セグメント(5DADH)をも知ることができるので，この前後のメモリ管理情報と共にメモリ・ダンプして確認します(⑨)．

そして，次にロードされるべきプログラムの PSP のダンプ・リストを見ます(⑩)．この PSP の内容からプロセス5の環境セグメント(5E9EH)を知り，その前後のメモリ管理情報やプロセス5の PSP を確認し，このプロセス5の PSP から中断処理アドレス(5DB7：013CH)とエラー処理アドレス(5DB7：04EBH)を知ることができます(⑪)．

プロセスの終了

プロセスの終了とは，子プロセスから親プロセスへ

戻ることを意味します．プロセスの終了には，正常な終了と異常が起こった場合の終了と2通りの終了があります(**表3-4**)．

さらに正常終了には，プログラムをメモリに残さないで終了するもの(ファンクション・リクエスト 4CH)と，プログラムをメモリに常駐したまま終了するもの(ファンクション・リクエスト 31H)の2通りがあります．

また，異常終了にも，ctrl-C による終了(INT 23H)，およびハード・エラーによる異常終了(INT 24H)があります．ハード・エラーとは，デバイス・レベルでのエラーのことであり，そのエラー情報はデバイス・ドライバから返されます．

正常終了の場合は，EXEC システム・コールが実行された時点で，割り込みベクタ・テーブルから PSP にコピーされている ctrl-C による中断アドレス，およびハード・エラー処理アドレスが PSP から元のベクタ・テーブルに戻され，親プロセスでの割り込みテーブルに復帰させるとともに INT 22H のベクタ・テーブルに書かれてある終了アドレスに復帰します．

また，子プロセスで INT 23H や INT 24H のベクタ・テーブルを書き替えていなければ(これらの処理ルーチンをもっていなければ)，ctrl-C による中断やハード・エラーが起こった時点で，親プロセスの各々の処理ルーチンに飛ぶことになり，やはり子プロセスは終了して親プロセスに戻ることになります．

親プロセスでは，システム・コールのファンクション 4DH により，子プロセスがどのような状態で終了したのかを，そのリターン・コードで知ることができます．

PSP(Program Segment Prefix)

プログラム(プロセス)が実行されるとき，プログラム自身はメモリの TPA(Transient Program Area)にロードされますが，このプログラムがロードされるべき空き領域のことをプログラム・セグメントと呼びます［**図3-2**(f)参照］．

このプログラム・セグメントの先頭 256 バイトには，MS-DOS とユーザ・プログラムとのインターフェースを取る領域が確保されますが，このインターフェース領域のことを PSP と呼びます．

ここには，親プロセス(command.com)から子プロセスへ渡されるパラメータの文字列や各種のアドレス，FCB(File Control Block)，DTA(Data Transfer Address)，あるいは MS-DOS のワーク・エリアなどが含まれます．

● PSP の構成

この PSP の構成は**図3-8** のようになっており，プログラム・セグメントのオフセット 0000H～0100H の領域が PSP 領域として確保されます．**リスト3-4** は，デバッガ symdeb を用いて masm.exe にパラメータを並べて起動した場合の PSP のダンプ・リストを示した

〔表3-4〕プロセス終了の種類

終了状態	終了原因
正常終了	通常終了：ファンクション・リクエスト 4CH
	常駐終了：ファンクション・リクエスト 31H
異常終了	ctrl-C 入力：INT 23H
	ハード・エラー：INT 24H

〔図3-8〕PSP の構成

〔リスト3-4〕PSPの内容

```
R>symdeb ⏎ …デバッガの起動
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-n masm.exe a:test1,b:test2,c:test3,d:test4 ⏎ …ターゲットとしてmasm.exeにパラメータを指定してロード
-l ⏎
-r ⏎ …各レジスタの確認
AX=0000  BX=0001  CX=AF33  DX=0000  SP=0080  BP=0000  SI=0000  DI=0000
DS=53CC  ES=53CC  SS=6FBE  CS=6E6C  IP=0010   NV UP EI PL NZ NA PO NC
6E6C:0010 8CC0          MOV     AX,ES     絶対アドレス9FFFFHまでのメモリ割り当て      プログラム終了
-d 0 ff ⏎…PSPのメモリ・ダンプ            ハード・エラー処理                          中断処理
53CC:0000  CD 20 00 A0 00 9A F0 FE-1D F0 28 09 E5 4A C5 09   M ...p~.p(.eJE.
53CC:0010  E5 4A F0 08 E5 4A D5 4A-06 07 01 00 02 FF FF FF   eJp.eJUJ........   環境セグメント
53CC:0020  FF FF FF FF FF FF FF FF-FF FF FF FF C2 53 A0 8D   ............BS .
53CC:0030  E5 4A 14 00 18 00 CC 53-FF FF FF FF 01 00 00 00   eJ....LS........
53CC:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
53CC:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 01 54 45 53   M!K.........TES    第1FCB
53CC:0060  54 31 20 20 20 20 20 20-00 00 00 00 02 54 45 53   T1      ....TES    第2FCB
53CC:0070  54 32 20 20 20 20 20 20-00 00 00 00 00 00 00 00   T2      ........
53CC:0080  29 20 4D 41 53 4D 2E 45-58 45 00 61 3A 74 65 73   ) MASM.EXE.a:tes
53CC:0090  74 31 2C 62 3A 74 65 73-74 32 2C 63 3A 74 65 73   t1,b:test2,c:tes
53CC:00A0  74 33 2C 64 3A 74 65 73-74 34 0D 00 00 00 00 00   t3,d:test4......
53CC:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................   パラメータ文字列および
53CC:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................   デフォルトDTA
53CC:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
53CC:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
53CC:00F0  00 00 00 00 00 00 00 00-00 00 00 EA 0F 56 E5 4A   ...........j.VeJ
-u ds:0 ⏎        パラメータの文字数
53CC:0000 CD20          INT     20  ←①
          PSPのオフセット0000Hの逆アセンブル

-u ds:5 ⏎ …PSPのオフセット0005Hの逆アセンブル
53CC:0005 9AF0FE1DF0    CALL    F01D:FEF0  ←②
          .
          .
-u ds:50 ⏎ …PSPのオフセット0050Hの逆アセンブル
53CC:0050 CD21          INT     21  }
53CC:0052 CB            RETF        } ③
          .
          .
-q ⏎
R>
```

ものです．

◆ **オフセット00H〜01H**

INT 20H(プログラム終了のシステム・コール)の機械語コード(CDH, 20H)が入っています(**リスト3-4**の①)．これはCP/Mが，

JMP　　0H

でプログラムを終了していたものとの互換性を考えて設けられています．

この方法によるプログラム終了は，後述のCOMモデルのプログラムでは，CS(コード・セグメント)とDS(データ・セグメント)が同じセグメントを指しているため簡単に行うことができます．

しかし，EXEモデルのプログラムでは，CS(コード・セグメント)がPSPのセグメントを指していないため，このPSP内のアドレスにジャンプするには，スタック内の戻り番地を変えるなどして，CS(コード・セグメント)の内容をPSPのセグメントに合わせなければならなくなります．

また，今後のMS-DOSのバージョン・アップやOS/2への移行などを考えると互換性の面で問題があり，ファンクション・リクエスト4CHによるプログラム終了を用いるべきです．

◆ **オフセット02H〜03H**

ここには使用可能メモリの最上位のセグメント・アドレス+1が入っています．同リストの例ではA000Hになっていますので，絶対アドレスの9FFFFHまでがMS-DOS(正確にはこの時点ではmasm.exeに割り当てられている)の使用可能メモリであることがわかります．

◆ **オフセット05H〜09H**

これもCP/Mとの互換性を考えて用意されたもので，この部分にはMS-DOSへのFAR CALL命令が入っています．

CP/M互換のシステム・コールでは，CLレジスタに

ファンクション番号をセットして，このアドレス0005HをNEAR CALLします．

しかし，この方法も今となっては過去の遺物といえる方法となりますので，今後の互換性を考えると使用しないほうが賢明といえます(**リスト3-4**の②)．

◆ オフセット0AH～0DH

この部分には，EXECシステム・コールにより子プロセスが起動された時点で，その子プロセスの終了アドレスが入ります．

MS-DOSは，親プロセスのEXECシステム・コールの次の命令のアドレスをINT 22Hの割り込みベクタ・テーブルに格納し，そのあとINT 22Hの割り込みベクタのコピーをこのフィールドにセットします．

◆ オフセット0EH～11H

この部分には，EXECシステム・コールによりプロセスが起動された時点で，INT 23Hの割り込みベクタ・テーブルがコピーされます．すなわち，この部分には親プロセスのctrl-Cによるプログラムの中断処理ルーチンのアドレスが入ることになります．

◆ オフセット12H～15H

オフセット0EH～11Hの4バイトと同様に，親プロセスのINT 24H(ハード・エラー)の割り込みベクタのコピーが入っています．

◆ オフセット2CH～2DH

この部分には，プロセスの環境のセグメント・アドレスが入っています．環境については，後述の環境の項を参照してください．

◆ オフセット50H～52H

INT 21HとRETFの機械語コードが入っています(**リスト3-4**の③)．したがって，第6章で紹介するシステム・コールを利用する際に，このアドレスに対して，FAR CALLしてもさしつかえないことになります．

◆ オフセット5CH(第1 FCB)，6CH(第2 FCB)

CP/M互換のために用意されたテーブルで，ディスク・アクセスに関するパラメータがセットされています．

これらのフィールドは十数バイトしかありませんが，実際にFCBとして使われるためには後述のように37バイトが必要になります．したがって，これを利用するには，ユーザ・プログラム内の作業領域にコピーしてから利用することになります．

◆ オフセット80H～FFH

PSPのオフセット80H～FFHの領域には，コマンドに渡されたパラメータの文字列が入ります．この場合のフォーマットは，80Hには文字数が入り，その後に文字列が続きます．文字列の最後にはデリミタとして0DHが入りますが，この0DHは文字数には数えられていません．

また，このエリアはデフォルトのDTA(FCBによってファイルをアクセスする場合のワーク・エリア)としても使用されます．DTAに関しては，第4章のFCBの項および第6章の該当するシステム・コールのところで解説します．

環境変数と環境

環境変数とは，setコマンドやpathコマンドにおいて定義された文字列のことをいいます．そして，これらの文字列(環境変数)を集めたものを"環境"と呼んでいます．

これらの文字列は，ASCIZ文字列と呼ばれる，ASCIIコード文字列にターミネータとして00Hを付けたもので表しています．また，この環境の最後には，さらに00Hをもう一つ付けて表現することになっています．

最近のコンパイラやエディタなど多くのプログラム開発ユーティリティは，この環境変数をうまく利用してライブラリの入っているディレクトリや，ヘルプ・ファイルのあるディレクトリを指定するなど，そのシステムに合ったプログラム開発環境を整えるようになってきています．

PSPのオフセット2CHにある環境アドレスとは，この環境文字列の置かれている先頭アドレスを指しています．このアドレスは，セグメント・アドレスになっていて，必ずオフセット0000Hから始まることになります．

そして，この環境は親プロセスから子プロセスの環境変数領域へコピーされて継承されます．また，この環境の継承は親プロセスから子プロセスへの一方通行なので，子プロセスで環境変数を変更しても親プロセスのもつ環境は保存されます．

ver.3.30より以前のバージョンでは，環境変数領域のサイズは固定長になっていて，子プロセスは親プロセスよりも大きな環境変数領域(16バイト・パラグラフ単位)を取ることはできませんでした．しかし，ver.3.30では，環境変数領域のサイズが可変長となり，command.comがMS-DOSから割り当てられたメモリ領域に環境を保存します．そして，その環境を子プロセスにコピーしているため，環境変数のサイズに制限がなくなっています．

【環境の実行サンプル】

リスト3-5は，環境変数や環境領域を調べるための実行例です．

① まず，親プロセス(command.com)のsetコマンドで環境変数の確認をしておく．

② 次に，子プロセスである symdeb.exe で，さらに孫プロセスに当たる command.com をロードする．

③ d コマンドを用いて孫プロセスの PSP を確認する．ここで，孫プロセスの PSP のオフセット 2CH から，この孫プロセスの環境セグメントを知ることができる．

④ そして，この環境セグメントの内容をデバッガ symdeb の d コマンドで確認する．すると，たとえばオフセット 29H には 00H が入っていて，これが環境変数(文字列)のターミネータになっていることがわかる．また，オフセット 82H を見れば，環境変数領域のターミネータとして，二つの 00H が並んでいることもわかる．

⑤ 次に，デバッガ symdeb の g コマンドで孫プロセ

〔リスト3-5〕子プロセスと環境領域 ①

```
R>set ⏎ … 親プロセスの環境を確認 ①
PATH=R:¥;H:¥BIN;H:¥MSC¥EXE;H:¥COM3;H:¥BAT  ┐
COMSPEC=R:¥COMMAND.COM                      │
HELP=H:¥BIN                                 ├ 環境
PROFILE=H:¥BIN                              │
INCLUDE=H:¥MSC¥INCLUDE¥                     │
MIMACRO=H:¥BIN                              ┘

R>symdeb command.com ⏎ … 子プロセス (symdeb) によって孫プロセス (command.com) をロード ②
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-d 0 ⏎ … 孫プロセス(command.com) の PSP ③
53CD:0000  CD 20 00 A0 00 9A F0 FE-1D F0 28 09 E5 4A C5 09   M . ..p~.p(.eJE.
53CD:0010  E5 4A F0 08 E5 4A D5 4A-06 07 01 00 02 FF FF FF   eJp.eJUJ........
53CD:0020  FF FF FF FF FF FF FF FF-FF FF FF FF C2 53 A2 8D   ............BS".
53CD:0030  E5 4A 14 00 18 00 CD 53-FF FF FF FF 01 00 00 00   eJ....MS........
53CD:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
53CD:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20   M!K.........
53CD:0060  20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20           .....
53CD:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00           ........
-d 53c2:0 1100 ⏎ … 孫プロセスの環境を確認 ④        └ 孫プロセスの環境セグメント
53C2:0000  50 41 54 48 3D 52 3A 5C-3B 48 3A 5C 42 49 4E 3B   PATH=R:¥;H:¥BIN;
53C2:0010  48 3A 5C 4D 53 43 5C 45-58 45 3B 48 3A 5C 43 4F   H:¥MSC¥EXE;H:¥CO
53C2:0020  4D 33 3B 48 3A 5C 42 41-54 00 43 4F 4D 53 50 45   M3;H:¥BAT.COMSPE
53C2:0030  43 3D 52 3A 5C 43 4F 4D-4D 41 4E 44 2E 43 4F 4D   C=R:¥COMMAND.COM
53C2:0040  00 48 45 4C 50 3D 48 3A-5C 42 49 4E 00 50 52 4F   .HELP=H:¥BIN.PRO
53C2:0050  46 49 4C 45 3D 48 3A 5C-42 49 4E 00 49 4E 43 4C   FILE=H:¥BIN.INCL
53C2:0060  55 44 45 3D 48 3A 5C 4D-53 43 5C 49 4E 43 4C 55   UDE=H:¥MSC¥INCLU
53C2:0070  44 45 5C 00 4D 49 4D 41-43 52 4F 3D 48 3A 5C 42   DE¥.MIMACRO=H:¥B
53C2:0080  49 4E 00 00 01 00 43 4F-4D 4D 41 4E 44 2E 43 4F   IN....COMMAND.CO
53C2:0090  4D 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   M...............
53C2:00A0  5A CD 53 33 4C 00 00 00-00 00 00 00 00 00 00 00   ZMS3L...........
53C2:00B0  CD 20 00 A0 00 9A F0 FE-1D F0 28 09 E5 4A C5 09   M . ..p~.p(.eJE.
53C2:00C0  E5 4A F0 08 E5 4A D5 4A-06 07 01 00 02 FF FF FF   eJp.eJUJ........
53C2:00D0  FF FF FF FF FF FF FF FF-FF FF FF FF C2 53 A2 8D   ............BS".
53C2:00E0  E5 4A 14 00 18 00 CD 53-FF FF FF FF 01 00 00 00   eJ....MS........
53C2:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-g ⏎ … 孫プロセスの実行 ⑤                   └ パラメータ文字列のターミネータ
                    └ 環境領域のターミネータ
Command バージョン 3.30

R>set test=abc ⏎ … 新しい環境変数の定義 ⑥

R>symdeb ⏎ … ひ孫プロセスの起動 ⑦
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-d 53c2:0 1100 ⏎ … 孫プロセスの環境の確認 ⑧
53C2:0000  50 41 54 48 3D 52 3A 5C-3B 48 3A 5C 42 49 4E 3B   PATH=R:¥;H:¥BIN;   ┐
53C2:0010  48 3A 5C 4D 53 43 5C 45-58 45 3B 48 3A 5C 43 4F   H:¥MSC¥EXE;H:¥CO   │
53C2:0020  4D 33 3B 48 3A 5C 42 41-54 00 43 4F 4D 53 50 45   M3;H:¥BAT.COMSPE   │
53C2:0030  43 3D 52 3A 5C 43 4F 4D-4D 41 4E 44 2E 43 4F 4D   C=R:¥COMMAND.COM   │
53C2:0040  00 48 45 4C 50 3D 48 3A-5C 42 49 4E 00 50 52 4F   .HELP=H:¥BIN.PRO   ├ 環境領域は変わらない⑨
53C2:0050  46 49 4C 45 3D 48 3A 5C-42 49 4E 00 49 4E 43 4C   FILE=H:¥BIN.INCL   │
53C2:0060  55 44 45 3D 48 3A 5C 4D-53 43 5C 49 4E 43 4C 55   UDE=H:¥MSC¥INCLU   │
53C2:0070  44 45 5C 00 4D 49 4D 41-43 52 4F 3D 48 3A 5C 42   DE¥.MIMACRO=H:¥B   │
53C2:0080  49 4E 00 00 01 00 43 4F-4D 4D 41 4E 44 2E 43 4F   IN....COMMAND.CO   ┘
53C2:0090  4D 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   M...............
53C2:00A0  4D CD 53 DB 00 00 00 00-00 00 00 00 00 00 00 00   MMS[............
53C2:00B0  CD 20 00 A0 00 9A F0 FE-1D F0 88 02 CD 53 C5 09   M . ..p~.p..MSE.
```

ス(command.com)を起動する.

⑥ そして，孫プロセスであるcommand.comのsetコマンドで，新しい環境変数(文字列)をセットする.

⑦ 次に，まえに調べておいた孫プロセス(command.com)の環境セグメントをチェックするために，さらにこの子プロセスとして孫プロセス(symdeb.exe)をロードする.

⑧ ここで，dコマンドを用いて孫プロセス(command.com)の環境を確認する.

⑨ ver.3.30より以前のバージョンでは，この環境領域に新しい環境変数が追加されていたが，ver.3.30では，新たに設定した環境は見あたらない.

⑩ そこで，ひ孫プロセス(symdeb.exe)が新たに起動するプロセスのために用意したPSPを調べ，新しい環

〔リスト3-5〕子プロセスと環境領域 ②

```
53C2:00C0  E5 4A F0 08 E5 4A CD 53-06 07 01 00 02 FF FF FF  eJp.eJMS........
53C2:00D0  FF FF FF FF FF FF FF FF-FF FF FF FF C2 53 75 07  ............BSu.
53C2:00E0  CD 53 14 00 18 00 CD 53-FF FF FF FF 01 00 00 00  MS...MS.........
53C2:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
-d 0 ⏎ …symdeb (ひ孫プロセス) が新しいプロセスのために用意しているPSPの確認 ⑩
5DAC:0000  CD 20 00 A0 00 9A 43 A0-07 F6 28 09 CF 54 C5 09  M . ..C .v(.OTE.
5DAC:0010  CF 54 F0 08 CF 54 CD 53-06 07 01 00 02 FF FF FF  OTp.OTMS........
5DAC:0020  FF FF FF FF FF FF FF FF-FF FF FF FF B4 54 75 07  ............4Tu.
5DAC:0030  CD 53 14 00 18 00 AC 5D-FF FF FF FF 01 00 00 00  MS....,]........
5DAC:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
5DAC:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 20 20 20  M!K.........
5DAC:0060  20 20 20 20 20 20 20 20-00 00 00 00 20 20 20       .....
5DAC:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00          ........
-d 54b4:0 1100 ⏎ … 新しい環境の確認 ⑪                 —新しい環境セグメント
54B4:0000  50 41 54 48 3D 52 3A 5C-3B 48 3A 5C 42 49 4E 3B  PATH=R:¥;H:¥BIN;
54B4:0010  48 3A 5C 4D 53 43 5C 45-58 45 3B 48 3A 5C 43 4F  H:¥MSC¥EXE;H:¥CO
54B4:0020  4D 33 3B 48 3A 5C 42 41-54 00 43 4F 4D 53 50 45  M3;H:¥BAT.COMSPE
54B4:0030  43 3D 52 3A 5C 43 4F 4D-4D 41 4E 44 2E 43 4F 4D  C=R:¥COMMAND.COM
54B4:0040  00 48 45 4C 50 3D 48 3A-5C 42 49 4E 00 50 52 4F  .HELP=H:¥BIN.PRO
54B4:0050  46 49 4C 45 3D 48 3A 5C-42 49 4E 00 49 4E 43 4C  FILE=H:¥BIN.INCL
54B4:0060  55 44 45 3D 48 3A 5C 4D-53 43 5C 49 4E 43 4C 55  UDE=H:¥MSC¥INCLU
54B4:0070  44 45 5C 00 4D 49 4D 41-43 52 4F 3D 48 3A 5C 42  DE¥.MIMACRO=H:¥B
54B4:0080  49 4E 00 54 45 53 54 3D-61 62 63 00 00 01 00 52  IN.TEST=abc....R
54B4:0090  3A 5C 57 4B 5C 53 59 4D-44 45 42 2E 45 58 45 00  :¥WK¥SYMDEB.EXE.
54B4:00A0  4D BF 54 EC 08 04 26 A2-0E 00 1E 50 56 B8 00 63  M?Tl..&"...PV8.c
54B4:00B0  CD 20 00 A0 00 9A F0 FE-1D F0 2F 01 CD 53 3C 01  M . ..p~.p/.MS<.
54B4:00C0  CD 53 EB 04 CD 53 CD 53-06 07 01 00 02 FF FF FF  MSk.MSMS........
54B4:00D0  FF FF FF FF FF FF FF FF-FF FF FF FF B4 54 8C 8D  ............4T..
54B4:00E0  CF 54 14 00 18 00 BF 54-FF FF FF FF 01 00 00 00  OT....?T........
54B4:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
-q ⏎ … 孫プロセス (command.com) に戻る ⑫     —新しい環境変数

R>set test=abcdefghijklmnopqrstuvwxyz ⏎ … 環境変数に新しいパラメータを設定 ⑬

R>symdeb ⏎ … 再びひ孫プロセスを起動 ⑭
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-d 0 ⏎ … PSPを調べ環境セグメントを確認 ⑮
5DAF:0000  CD 20 00 A0 00 9A 40 A0-08 F6 28 09 D2 54 C5 09  M . ..@ .v(.RTE.
5DAF:0010  D2 54 F0 08 D2 54 CD 53-06 07 01 00 02 FF FF FF  RTp.RTMS........
5DAF:0020  FF FF FF FF FF FF FF FF-FF FF FF FF B5 54 75 07  ............5Tu.
5DAF:0030  CD 53 14 00 18 00 AF 5D-FF FF FF FF 01 00 00 00  MS..../]........
5DAF:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
5DAF:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 20 20 20  M!K.........
5DAF:0060  20 20 20 20 20 20 20 20-00 00 00 00 20 20 20       .....
5DAF:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00          ........
-d 54b5:0 1100 ⏎ … 環境の確認 ⑯                      —環境セグメント
54B5:0000  50 41 54 48 3D 52 3A 5C-3B 48 3A 5C 42 49 4E 3B  PATH=R:¥;H:¥BIN;
54B5:0010  48 3A 5C 4D 53 43 5C 45-58 45 3B 48 3A 5C 43 4F  H:¥MSC¥EXE;H:¥CO
54B5:0020  4D 33 3B 48 3A 5C 42 41-54 00 43 4F 4D 53 50 45  M3;H:¥BAT.COMSPE
54B5:0030  43 3D 52 3A 5C 43 4F 4D-4D 41 4E 44 2E 43 4F 4D  C=R:¥COMMAND.COM
54B5:0040  00 48 45 4C 50 3D 48 3A-5C 42 49 4E 00 50 52 4F  .HELP=H:¥BIN.PRO
54B5:0050  46 49 4C 45 3D 48 3A 5C-42 49 4E 00 49 4E 43 4C  FILE=H:¥BIN.INCL
54B5:0060  55 44 45 3D 48 3A 5C 4D-53 43 5C 49 4E 43 4C 55  UDE=H:¥MSC¥INCLU
54B5:0070  44 45 5C 00 4D 49 4D 41-43 52 4F 3D 48 3A 5C 42  DE¥.MIMACRO=H:¥B
54B5:0080  49 4E 00 54 45 53 54 3D-61 62 63 64 65 66 67 68  IN.TEST=abcdefgh
54B5:0090  69 6A 6B 6C 6D 6E 6F 70-71 72 73 74 75 76 77 78  ijklmnopqrstuvwx
54B5:00A0  79 7A 00 00 01 00 52 3A-5C 57 4B 5C 53 59 4D 44  yz....R:¥WK¥SYMD
54B5:00B0  45 42 2E 45 58 45 00 53-FF FF FF FF FF FF FF FF  EB.EXE.S........
54B5:00C0  4D C2 54 EC 08 FF FF FF-FF FF FF FF B4 54 A6 8D  MBTl........4T&.

                                          —新しいパラメータ
```

境の格納されているセグメントを調べる．

⑪ そして，dコマンドを用いて新しい環境の確認を行う．すると，新たに設定した環境変数は正常に登録されている．

⑫ ここで，一度孫プロセス(command.com)に戻る．

⑬ 次に，setコマンドを用いてすでに定義されている環境変数(TEST)に新しいパラメータ(以前よりも長い)を設定する．

⑭ そして，再びひ孫プロセス(symdeb.exe)を起動する．

⑮ PSPを調べて環境セグメント・アドレスを確認する．

⑯ そして，その環境セグメントをメモリ・ダンプする．すると，パラメータが以前よりも長い場合でも正

〔リスト3-5〕子プロセスと環境領域 ③

```
54B5:00D0  CD 20 00 A0 00 9A F0 FE-1D F0 2F 01 CD 53 3C 01   M . ..p~.p/.MS<.
54B5:00E0  CD 53 EB 04 CD 53 CD 53-06 07 01 00 02 FF FF FF   MSk.MSMS........
54B5:00F0  FF FF FF FF FF FF FF FF-FF FF FF FF B5 54 8C 8D   ............5T..
-q ⏎ … 孫プロセスに戻る⑰

R>set test=abcdefg ⏎ … パラメータを以前より短い文字列で設定⑱

R>symdeb ⏎ … 再びデバッガを起動⑲
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-d 0 ⏎ … PSPを調べ環境セグメントを確認⑳
5DAE:0000  CD 20 00 A0 00 9A 41 A0-07 F6 28 09 D1 54 C5 09   M . ..A .v(.QTE.
5DAE:0010  D1 54 F0 08 D1 54 CD 53-06 07 01 00 02 FF FF FF   QTp.QTMS........
5DAE:0020  FF FF FF FF FF FF FF FF-FF FF FF FF B5 54 75 07   ............5Tu.
5DAE:0030  CD 53 14 00 18 00 AE 5D-FF FF FF FF 01 00 00 00   MS....]........
5DAE:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
5DAE:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20   M!K.............
5DAE:0060  20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20   ................
5DAE:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00   ................
-d 54b5:0 1100 ⏎ … 環境の確認                          └環境セグメント
54B5:0000  50 41 54 48 3D 52 3A 5C-3B 48 3A 5C 42 49 4E 3B   PATH=R:¥;H:¥BIN;
54B5:0010  48 3A 5C 4D 53 43 5C 45-58 45 3B 48 3A 5C 43 4F   H:¥MSC¥EXE;H:¥CO
54B5:0020  4D 33 3B 48 3A 5C 42 41-54 00 43 4F 4D 53 50 45   M3;H:¥BAT.COMSPE
54B5:0030  43 3D 52 3A 5C 43 4F 4D-4D 41 4E 44 2E 43 4F 4D   C=R:¥COMMAND.COM
54B5:0040  00 48 45 4C 50 3D 48 3A-5C 42 49 4E 00 50 52 4F   .HELP=H:¥BIN.PRO
54B5:0050  46 49 4C 45 3D 48 3A 5C-42 49 4E 00 49 4E 43 4C   FILE=H:¥BIN.INCL
54B5:0060  55 44 45 3D 48 3A 5C 4D-53 43 5C 49 4E 43 4C 55   UDE=H:¥MSC¥INCLU
54B5:0070  44 45 5C 00 4D 49 4D 41-43 52 4F 3D 48 3A 5C 42   DE¥.MIMACRO=H:¥B
54B5:0080  49 4E 00 54 45 53 54 3D-61 62 63 64 65 66 67 00   IN.TEST=abcdefg.
54B5:0090  00 01 00 52 3A 5C 57 4B-5C 53 59 4D 44 45 42 2E   ...R:¥WK¥SYMDEB.
54B5:00A0  45 58 45 00 01 00 52 3A-5C 57 4B 5C 53 59 4D 44   EXE...R:¥WK¥SYMD
54B5:00B0  4D C1 54 EC 08 45 00 53-FF FF FF FF FF FF FF FF   MAT1.E.S........
54B5:00C0  CD 20 00 A0 00 9A F0 FE-1D F0 2F 01 CD 53 3C 01   M . ..p~.p/.MS<.
54B5:00D0  CD 53 EB 04 CD 53 CD 53-06 07 01 00 02 FF FF FF   MSk.MSMS........
54B5:00E0  FF FF FF FF FF FF FF FF-FF FF FF FF B5 54 8C 8D   ............5T..
54B5:00F0  D1 54 14 00 18 00 C1 54-FF FF FF FF 01 00 00 00   QT....AT........
-q ⏎ … 孫プロセスに戻る㉑                               └新しいパラメータ

R>set ⏎ … 環境の確認㉒
PATH=R:¥;H:¥BIN;H:¥MSC¥EXE;H:¥COM3;H:¥BAT  ┐
COMSPEC=R:¥COMMAND.COM                      │
HELP=H:¥BIN                                 │
PROFILE=H:¥BIN                              ├ 孫プロセスの環境
INCLUDE=H:¥MSC¥INCLUDE¥                     │
MIMACRO=H:¥BIN                              │
TEST=abcdefg ←--- 定義した環境              ┘

R>exit ⏎ … 子プロセスに戻る㉓

Program terminated normally (0)
-q ⏎ … 親プロセスに戻る㉔

R>set ⏎ … 環境の確認㉕
PATH=R:¥;H:¥BIN;H:¥MSC¥EXE;H:¥COM3;H:¥BAT  ┐
COMSPEC=R:¥COMMAND.COM                      │
HELP=H:¥BIN                                 ├ 親プロセスの環境
PROFILE=H:¥BIN                              │
INCLUDE=H:¥MSC¥INCLUDE¥                     │
MIMACRO=H:¥BIN                              ┘

R>
```

常に設定される.

⑰ ここで,孫プロセスに戻る.

⑱ set コマンドを用いて,環境変数に対して,新しいパラメータ(以前よりも短い)を設定する.

⑲ 再びび孫プロセス(symdeb.exe)を起動する.

⑳ PSPを調べて環境セグメントを確認する.

㉑ 孫プロセスに戻る.

㉒ set コマンドで孫プロセスの環境を確認する.

㉓ 子プロセス(symdeb.exe)に戻る.

㉔ 親プロセスに戻る.

㉕ set コマンドで親プロセスの環境を確認する.このように,親プロセスの環境は保存され,子プロセスで環境を変更しても親プロセスの環境は影響を受けない.

実行プログラムのメモリ・モデル

これまで述べてきたように,MS-DOSではプログラムが階層的な構造をしており,そのメモリ配置はプログラムがロードされるまで決定されず,プログラム自体はリロケータブル(再配置可能)な構造になっていなければなりません.

プログラム・サイズが64Kバイト以内であれば,このプログラムがロードされる際に四つのセグメント・レジスタに初期設定する値を変えるだけで,プログラムのリロケートは可能になります.

しかし,64Kバイトを越えるプログラムでは,論理的にいくつかのセグメントをもつことになり,プログラム内で外部のセグメントの参照が行われることになります(第2章MASM疑似命令参照).このような場合に,プログラムをリロケートするためには,プログラム内でセグメントを参照している命令のオペランドを書き替えてやらなければなりません.

MS-DOSでは,前者のような64Kバイト以内のプログラムの形式を"COMモデル",後者のように64Kバイト以上のプログラムで,外部のセグメント参照を行うプログラムの形式を"EXEモデル"と呼んで,それぞれ異なった管理方法を用いることにより,プログラムのリロケーションを実現しています.

● EXEモデル

MS-DOSが対象としている8086 CPUでは,1Mバイトのメモリ空間を64Kバイトごとのセグメントに区切って管理しています.また,アセンブラMASMやリンカLINKなども,このようなセグメント管理をサポートしています.

そして,これらのセグメントの参照や,セグメント間ジャンプのオペランドなどはLINKにその情報が引き渡されます.最終的に,これらの情報はLINKから出力される実行形式ファイル(".exe"ファイル)の中にリロケーション情報として引き渡され,セグメント参照のオペランドは,これらのプログラムが実際にメモリにロードされた時点で確立します.

〔図3-9〕EXEモデルのファイル構造

図3-9は,EXEモデルのファイル構造を示しています.EXEモデルのファイルでは,セグメントを参照しているオペランドの位置や個数,あるいは各レジスタに初期設定される値などの入ったヘッダ部分と,実際にユーザが意識しているプログラムやデータの入ったロード・モジュール部分に分けられます.

そして,ヘッダ部分はさらにリロケーション・テーブルの入っている部分(リロケート情報)と,これらのリロケートを行う際に必要な各種のパラメータ,あるいはIPやSPレジスタに初期設定されるべき値などの入ったコントロール情報に分けられます.**表3-5**はヘッダのフォーマットをまとめたものです.

◆ **オフセット00H~01H**

常に4D5AHの1ワードの値が入ります.これは,このファイルが有効なEXEファイルであることを示すためにLINKによって付けられたマークです.

◆ **オフセット02H~03H**

このファイルの最後のページに入っているバイト数が入ります.ここでページとは,512バイトの大きさのファイル・ブロックをいいます.

◆ **オフセット04H~05H**

このフィールドにはEXEファイルのページの大きさが入ります.このページの大きさとオフセット02H~03Hの最後のページに入っているバイト数とから,

〔表3-5〕ヘッダのコントロール情報

オフセット(16進)	内　　　容
00~01H	4DH, 5AH. 有効なEXEファイルであることを示すマーク
02~03H	最終のページ(512バイト単位)に入っているバイト数
04~05H	ページ(512バイト単位)の数
06~07H	リロケートする項目数
08~09H	ヘッダの大きさ(16バイト・パラグラフ)
0A~0BH	ロードされたプログラムの後に必要な16バイト・パラグラフの最小数
0C~0DH	ロードされたプログラムの後に必要な16バイト・パラグラフの最大数
0E~0FH	ロードされたプログラムのスタック・セグメントのオフセット
10~11H	SPレジスタに設定される値
12~13H	ファイル内のワード単位によるネガティブ・サム
14~15H	IPレジスタに設定される値
16~17H	ロードされたプログラムのコード・セグメントのオフセット
18~19H	最初のリロケーション項目のオフセット
1A~1BH	オーバレイ番号

リロケーション情報を含むファイル全体の大きさが計算できることになります．

◆ **オフセット 06H~07H**

リロケートすべき項目の数が入っています．

◆ **オフセット 08H~09H**

このフィールドにはヘッダの大きさが入っています．これとページ数や最後のページに入っているバイト数から，リロケートが終わったあとのロード・モジュールの大きさが計算できることになります．

◆ **オフセット 0AH~0BH**

ロードされたプログラムの後に必要な16バイト・パラグラフの最少数が入ります．

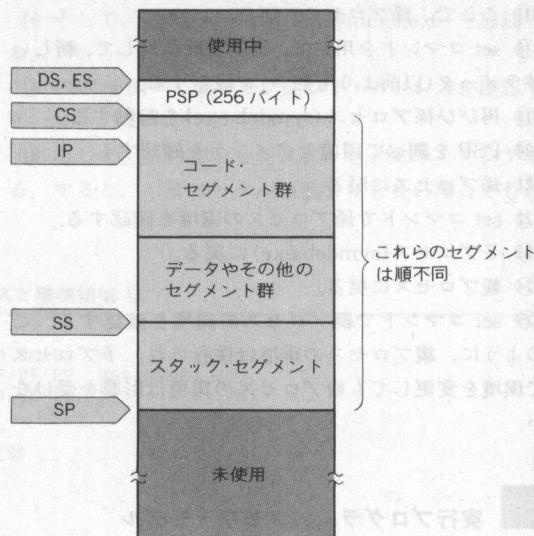
◆ **オフセット 0CH~0DH**

ロードされたプログラムの後に必要な16バイト・パラグラフの最大数が入ります．これらの必要とするメモリ(パラグラフ)の最少数や最大数の情報により，そのプログラムのメモリ内での配置が決定されます．

これらの値がともにゼロの場合は，プログラムはできるだけメモリの上位アドレスにロードされます．

◆ **オフセット 0EH~0FH**

ロード・モジュール内のスタック・セグメントの位置が入ります．この値はロード・モジュールの開始点からのパラグラフ数のオフセットで表されます．この値によりSS(スタック・セグメント)レジスタが初期設定されます．

〔図3-10〕EXEモデルにおける各レジスタの初期値

◆ **オフセット 10H~11H**

SP(スタック・ポインタ)レジスタに設定される初期値がセットされます．

◆ **オフセット 12H~13H**

ファイル内の全データをワード単位でチェックしたネガティブ・サムが入ります．この場合，オーバフローは無視されています．

◆ **オフセット 14H~15H**

IP(インストラクション・ポインタ)レジスタに初期設定されるべき値がセットされます．アセンブリ記述の場合は，ORGディレクティブで指定したアドレスがセットされることになります．

◆ **オフセット 16H~17H**

ロード・モジュール内のコード・セグメントの値が，ロード・モジュールの開始点からの16バイト・パラグラフのオフセット値で入ります．この値によりプログラム実行時のCS(コード・セグメント)レジスタが初期設定されます．

◆ **オフセット 18H~19H**

ファイル内の先頭のリロケーション項目のオフセット値が入ります．

◆ **オフセット 1AH~1BH**

オーバレイ番号が入ります．常駐部の場合は，ここにはゼロが入ります．

＊　　　＊

EXEモデルのプログラムがロードされた場合，各レジスタは図3-10のように初期設定されます．

〔リスト3-6〕EXEモデルのプログラム例

```
                ;*************************************************************
                ;
                ;  機    能:  EXEモデルのプログラミング例
                ;  生    成:  masm /ML exe;
                ;             link /NOI/MAP exe,,exe;
                ;
                ;*************************************************************
                        PAGE    ,132
                        PUBLIC  data1, data2, addr1, addr2, addr3, addr4
                        PUBLIC  start, sub1

0000            cseg1   SEGMENT 'CODE'                          ;コード・セグメント
                        ASSUME  CS:cseg1, DS:dseg1, ES:dseg2, SS:sseg

0000            start   PROC
0000  B8 ---- R         mov     ax, SEG data1
0003  8E D8             mov     ds, ax                          ;DSレジスタの初期化
0005  B8 ---- R         mov     ax, dseg2                       ;セグメント参照
0008  C4 3E 0004 R      les     di, addr2                       ;セグメント参照
000C  C4 3E 0008 R      les     di, addr3                       ;セグメント参照
0010  8B 1E 0000 R      mov     bx, data1
0014  8B 36 0002 R      mov     si, addr1
0018  B8 ---- R         mov     ax, dseg2
001B  8E C0             mov     es, ax                          ;ESレジスタの初期化
001D  26: 8B 3E 0002 R  mov     di, dseg2:addr4
0022  9A 0000 ---- R    call    FAR PTR sub1                    ;セグメント参照
0027  B4 4C             mov     ah, 4Ch
0029  B0 00             mov     al, 00h                         ;リターン・コード
002B  CD 21             int     21h                             ;プログラム終了
002D            start   ENDP
002D            cseg1   ENDS

0000            cseg2   SEGMENT 'CODE'                          ;コード・セグメント

0000            sub1    PROC    FAR
0000  CB                ret
0001            sub1    ENDP
0001            cseg2   ENDS

0000            sseg    SEGMENT STACK   'SATCK'                 ;スタック・セグメント
0000    0080[           DW      128 DUP (?)
          ????
              ]
0100            sseg    ENDS

0000            dseg1   SEGMENT 'DATA'                          ;データ・セグメント
0000  1111      data1   DW      1111h
0002  0000 R    addr1   DW      data1
0004  0000 ---- R addr2 DD      data1                           ;セグメント参照
0008  0000 ---- R addr3 DD      data2                           ;セグメント参照
000C            dseg1   ENDS

0000            dseg2   SEGMENT 'DATA'                          ;データ・セグメント
0000  2222      data2   DW      2222h
0002  0000 R    addr4   DW      data2
0004            dseg2   ENDS
                        END     start
```


【EXEモデルのサンプル・プログラム】

リスト3-6(exe.asm)は，EXEモデルのプログラミング例を示しています．同リストでは，複数の論理セグメントを配置して，多くのセグメント参照を行っています．通常のプログラム開発では，これらのセグメントは別々のファイル(モジュール)として記述され，リンク時に一つの実行モジュールとして生成されます．

さて，同リストのようなセグメント参照はEXEモデルのプログラムのみに許され，後述のCOMモデルのプログラムでは利用できません．なお，同リストでは，MAPファイルにおいてセグメント配置に加えてシンボル配置の状況も確認するため，すべてのシンボル(ラベル)をPUBLIC宣言しています．

リスト3-7はLINKから出力されたMAPファイル(exe.map)の内容です．このMAPファイルの内容から，各セグメントの配置は**図3-11**のように表すことができます．

リスト3-8は**リスト3-6**(exe.asm)のソース・ファイルをアセンブル/リンクした結果得られた実行モジュール(exe.exe)をDUMPコマンドを用いてファイル・ダンプしたもので，コントロール情報の確認を行っています．

〔リスト3-7〕exe.asm から得られた MAP ファイル

```
Start  Stop   Length Name               Class
00000H 00030H 00031H cseg1              CODE
00040H 00040H 00001H cseg2              CODE
00050H 0005BH 0000CH dseg1              DATA
00060H 00063H 00004H dseg2              DATA
00070H 0016FH 00100H sseg               SATCK

  Address         Publics by Name

 0005:0002        addr1
 0005:0004        addr2
 0005:0008        addr3
 0006:0002        addr4
 0005:0000        data1
 0006:0000        data2
 0000:0000        start
 0004:0000        sub1

  Address         Publics by Value

 0000:0000        start
 0004:0000        sub1
 0005:0000        data1
 0005:0002        addr1
 0005:0004        addr2
 0005:0008        addr3
 0006:0000        data2
 0006:0002        addr4

Program entry point at 0000:0000
```

〔図3-11〕テスト・プログラム(exe.exe)のセグメント配置

〔リスト3-8〕
exe.exe のファイル・ダンプ

```
                 最後のバイト    2ページ(1Kバイト)          0110Hバイト
            マーク                     6個  0200H開始          下位     SS
00000000  4D 5A 54 00 02 00 06 00-20 00 11 00 FF FF 06 00   MZT.............
00000010  00 01 50 F6 00 00 00 00-1E 00 00 00 01 00 01 00   ..P.............
00000020  00 00 06 00 00 00 19 00-00 00 25 00 00 00 06 00   ..........%.....
00000030  04 00 0A 00 04 00 00 00-00 00 00 00 00 00 00 00   ................
00000040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
          SP            IP      CS     001EHからリロケーション項目
               チェック・サム

000001F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000200 →B8 04 00 8E D8 B8 05 00-C4 3E 04 00 C4 3E 08 00   ｸ..借ｸ..ﾄ>..ﾄ>..
00000210  8B 1E 00 00 8B 36 02 00-B8 05 00 8E C0 26 8B 3E   .....6..ｸ..実&.>
00000220  02 00 9A 30 00 00 00 B4-4C B0 00 CD 21 00 00 00   ...0...ｴLｰ.ﾍ!...
00000230  CB 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ﾋ...............
00000240  11 11 00 00 00 00 04 00-00 00 05 00 00 00 00 00   ................
00000250  22 22 00 00 4E 42 30 30-CD 00 00 00 00 00 00 00   ""..NB00ﾍ.......
          ロード・モジュール開始
```

同リストの情報から，ファイル全体の大きさは1Kバイト(2ページ)であり，このうちヘッダ部分が512バイト(0200H)を占めていて，最後のページのバイト数から，実際のロード・モジュールの大きさは0054Hバイトであることがわかります.

また，SSレジスタには，プログラム・セグメント(ロードされたセグメント・アドレス)＋0006Hがセットされ，SPレジスタには0100Hが初期設定されます.同様にCSレジスタには，プログラム・セグメント＋0000Hがセットされ，IPレジスタには0000Hが初期設定されて，このアドレスから実行されることになります.

また，リロケーション項目は6個あり，その最初はヘッダ部分の1EHから始まっていることがわかります. 1EHからの4バイトの情報から，このセグメント値(0000H)にスタート・セグメント値(プログラム・セグメントのアドレス値)を加算します.この結果をロード・モジュール内のオフセット(0001H)にワード値としてセットします.

同様にこのヘッダのオフセット22Hの4バイトの情報からオフセット0006Hにもプログラム・セグメント値がセットされることになります．以下，同様の作業が6個のリロケーション項目に対して行われることによって，プログラムのロケートが可能になります.

このEXEモデルのメリットとしては，8086 CPUのアーキテクチャが忠実に反映されたものになっており，プログラムやデータをモジュールに分割して考えることができ，プログラム開発時の分担を行う場合や，プログラムの保守性の面でも利点があります.

また，反面ではその欠点として，構造が複雑でセグメント・レジスタの変更などの管理はユーザ・プログラム内で行わなければなりません．また，プログラム・

ファイルにリロケート情報が含まれているためにファイル・サイズも大きくなり，リロケートのための時間も必要なため，プログラムのロード時間も遅くなります．

● COMモデル

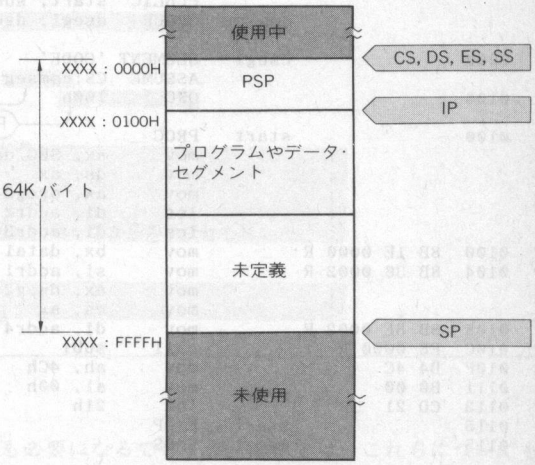
一般にアセンブリ・プログラムでは，コード部分が64Kバイト以上ものプログラムを作成するようなことはあまりありません．データ部分については，ユーザ・プログラム内でDS(データ・セグメント)レジスタを制御することによって，大きなデータを扱うことも可能となります．

そこで，この64Kバイトの制約を設けてEXEモデルの欠点を補い，コマンド・ファイルを実現しているものにCOMモデルがあります．COMモデルは，別名スモール・モデルとも呼ばれ，プログラムのコードやデータ，スタックは合わせて64Kバイト以下でなければなりません(同一のセグメントとして扱われるものでなければならない)．ただし，作業用データ領域は工夫しだい(プログラム内でDSを変更するなど)で64Kバイトに制約されることはありません．

また，プログラムはPSP領域を確保する必要があるため，

```
ORG     100H
```

で始めなければなりません．図3-12は，COMモデルのプログラムがロードされた場合の各レジスタの初期設定の状態を表しています．

〔図3-12〕COMモデルにおける各レジスタの初期値

● 8086 vs 68000(その2) ストリング命令 ●

8086では，ストリング命令やリピート命令が強化され，文字列の操作が高速に処理できるようになりました．

リストB(a)は，8086を用いて64Kバイト以上のメモリ・クリアを行うプログラム例で，8ビットCPUに比較してループ内の命令が簡潔に記述されていることがわかります．

しかし，同リスト(b)は68000の場合であり，8086よりももっと簡潔に記述することができています．そして，8086の場合は2バイトずつクリアしていきますが，68000では4バイトずつクリアしています．すなわち，ループの回数が半分ですむのです．

〔リストB〕メモリ・クリアの比較

```
        mov     bx,SEG_BGN   …セグメント初期値
        xor     ax,ax        …データ(0000H)
        xor     di,di
loop1:  mov     es,bx        } ポインタ初期値
        mov     cx,8000h
loop2:  rep     stosw        …メモリ・クリア
        inc     bx
        cmp     bx,SEG_END   } つぎのセグメント
        jne     loop1
```

(a) 8086の場合

```
        lea     MEM_BGN,a0     } カウンタ/ポインタの初期化
        moveq   #MEM_CNT,d0
        moveq   #0,d1          …データ
loop1   move.l  d1,(a0)+       …メモリ・クリア
        dbra    d0,loop1       …カウンタ更新
```

(b) 68000の場合

〔リスト3-9〕 COM モデルのプログラム例

```
                ;********************************************************************
                ;    機    能 :   COMモデルのプログラミング例
                ;    生    成 :   masm /ML com;
                ;                 link /NOI/MAP com,,com;
                ;                 exe2bin com.exe com.com
                ;
                ;********************************************************************
                        PAGE    ,132
                        PUBLIC  data1, data2, addr1, addr4
                        PUBLIC  start, sub1
                comseg  GROUP   dseg1, dseg2, cseg1, cseg2   ;セグメントのグループ化

0000            cseg1   SEGMENT 'CODE'                       ;コード・セグメント
                        ASSUME  CS:comseg, DS:comseg, ES:comseg, SS:comseg
0100                    ORG     100h

0100            start   PROC
                ;       mov     ax, SEG data1                ;セグメント参照
                ;       mov     ds, ax                       ;DSレジスタの初期化
                ;       mov     ax, dseg2                    ;セグメント参照
                ;       les     di, addr2                    ;セグメント参照
                ;       les     di, addr3                    ;セグメント参照
0100  8B 1E 0000 R      mov     bx, data1
0104  8B 36 0002 R      mov     si, addr1
                ;       mov     ax, dseg2
                ;       mov     es, ax                       ;ESレジスタの初期化
0108  8B 3E 0002 R      mov     di, addr4
010C  E8 0000 R         call    sub1
010F  B4 4C             mov     ah, 4Ch
0111  B0 00             mov     al, 00h                      ;リターン・コード
0113  CD 21             int     21h                          ;プログラム終了
0115            start   ENDP
0115            cseg1   ENDS

0000            cseg2   SEGMENT 'CODE'                       ;コード・セグメント
                        ASSUME  CS:comseg
0000            sub1    PROC    NEAR
0000  C3                ret
0001            sub1    ENDP
0001            cseg2   ENDS

                ;  sseg    SEGMENT STACK    'SATCK'          ;スタック・セグメント
                ;          DW      128 DUP (?)
                ;  sseg    ENDS

0000            dseg1   SEGMENT 'DATA'                       ;データ・セグメント
0000  1111      data1   DW      1111h
0002  0000 R    addr1   DW      data1
                ;  addr2   DD      data1                     ;セグメント参照
                ;  addr3   DD      data2                     ;セグメント参照
0004            dseg1   ENDS

0000            dseg2   SEGMENT 'DATA'                       ;データ・セグメント
0000  2222      data2   DW      2222h
0002  0000 R    addr4   DW      data2
0004            dseg2   ENDS
                        END     start
```


【COM モデルのサンプル・プログラム】

リスト3-9(com.asm)は，**リスト3-6**(exe.asm)のプログラムを COM モデル用に変えた例です．

COM モデルでは，セグメントの参照を行うことはできません．なぜなら，COM モデルでは 64 K バイト(同一のセグメント)以下のコードやデータを扱うため，セグメントを固定しているからです．また，**図3-12** が示すようにスタック・セグメントも他のセグメントと同様に固定されているため，論理的なスタック・セグメントの記述も許されません．

同リストでは，複数の論理セグメントを GROUP ディレクティブによって一つのセグメントにまとめています．これにより，COM モデルの場合でもプログラムを複数のモジュールで構成することも可能です．

リスト3-10 は，**リスト3-9**(com.asm)をアセンブル/リンクした結果得られた MAP ファイルの内容です．ここでは，GROUP ディレクティブによってセグメントが一つにまとめられているため，シンボルのオフセットは同一のセグメント・ベースからのオフセットが与えられているのが確認できます．

〔リスト3-10〕
com.asm から得られたMAPファイル

```
LINK : warning L4021: no stack segment

 Start  Stop   Length Name                          Class
 00000H 00117H 00118H cseg1                         CODE
 00120H 00120H 00001H cseg2                         CODE
 00130H 00133H 00004H dseg1                         DATA
 00140H 00143H 00004H dseg2                         DATA

 Origin   Group
 0000:0   comseg

  Address         Publics by Name

 0000:0132       addr1
 0000:0142       addr4
 0000:0130       data1
 0000:0140       data2
 0000:0100       start
 0000:0120       sub1

  Address         Publics by Value

 0000:0100       start
 0000:0120       sub1
 0000:0130       data1
 0000:0132       addr1
 0000:0140       data2
 0000:0142       addr4

Program entry point at 0000:0100
```

1個のセグメントにまとめられる

グループ名

同一セグメント・ベースからのオフセット

＊　　　　＊

この章では，MS-DOSのメモリ配置とメモリ・モデルを解説しました．MS-DOS上でプログラム開発を行う際に，メモリ・モデルに関する知識は必要不可欠なものとなります．

MS-DOSのメモリ・モデルの解説は他書においても数多く取り上げられていますが，ポイントは8086 CPUのセグメント管理の方法にあるといえるでしょう．すなわち，セグメント定義の方法や外部参照，セグメントのグループ化といったセグメントを操作するための知識は，しっかりと身につけておく必要があります．

また，ある種のアプリケーションでは，メモリ管理情報やPSP内の情報，およびメモリ配置に関する知識も必要になるでしょう．本章では，これらについても，実行例やアクセス・プログラムの実例を示したため，大いに参考になるものと思います．

つぎに，プログラミング環境を整えるには，config.sysファイルとシステム構築用のコマンドを上手に活用したいものです．

本章でも述べたように，MS-DOSではバッファの設定によって処理速度が大幅に改善されることがあります．また，プログラミング環境を整えるには，デバイス・ドライバの追加や環境変数の利用も重要な要素になってきます．

本章で解説した知識を活用することによって，効率的で快適なプログラミング環境を得ることができます．

第4章 MS-DOSのファイル・アクセス

ディレクトリとファイル・ハンドルとFCB

前章ではMS-DOSの内部構造について，システムのブートストラップからプログラムのロード/実行,環境などについて解説しました．ここでは，MS-DOSで扱われるファイルの構造とそのアクセス方法について解説していくことにします．

MS-DOS上で動作するアプリケーション・プログラムの開発に当たっては，このファイルの扱いについて習熟しておくことがたいせつです．

4-1 ファイル構造

MS-DOSでは,ファイルの管理を二つのテーブルを用いることによって実現しています．

(1) ディレクトリ

ファイルの名前や属性とその入口などのテーブル

(2) FAT(File Allocation Table)

実際にファイルの内容が置かれているディスク上の場所を管理するテーブル

これは，マイクロソフト社のBASICにおけるファイルの管理方法と非常に類似した管理方法です．

また，MS-DOSではディスクの管理の単位として，やはりBASICでお馴染みのクラスタと呼ばれる，ディスク上のデータ領域に付けられた論理的なレコード番号を用いて管理しています．

クラスタとは，セクタをいくつか集めたブロックをいい，クラスタとセクタの関係はディスクの種類によって異なります．**表4-1**は，MS-DOSで使われる主なディスク・フォーマットの一覧です．

ディスク上の領域

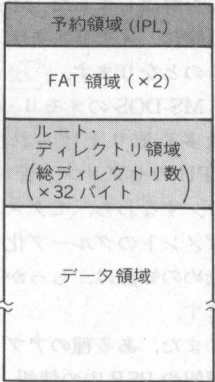
MS-DOSで扱うディスクは，通常は図4-1のように4個のブロックで構成されています(デバイス・ドライ

〔図4-1〕
ディスク上の領域の割り当て

予約領域 (IPL)

FAT 領域 (×2)

ルート・
ディレクトリ領域
(総ディレクトリ数
×32バイト)

データ領域

〔表4-1〕
MS-DOSのおもな ディスク・フォーマット

	5インチ・ディスク								8インチ・ディスク		
トラック数	80	80	80	80	40	40	40	40	77	77	77
セクタ/トラック	9	9	8	8	9	9	8	8	26	26	8
サイド数	1	2	1	2	1	2	1	2	1	1	2
セクタ・サイズ	512	512	512	512	512	512	512	512	128	128	1024
ディスク容量(KB)	360	720	320	640	180	360	160	320	250	250	1230
ディレクトリ数	112	112	112	112	64	112	64	112	68	68	192
セクタ/クラスタ	2	2	2	2	1	2	1	2	4	4	1
予約セクタ	1	1	1	1	1	1	1	1	1	4	1
FAT数	2	2	2	2	2	2	2	2	2	2	2
セクタ/FAT	2	3	1	2	2	2	1	1	6	6	2
FAT-ID	F8	F9	FA	FB	FC	FD	FE	FF	FE	FD	FE

バのBPBテーブルで規定される．第8章参照)．**リスト4-1** はデバッガ symdeb を用いて，1Mバイト・フォーマットのフロッピ・ディスクの各領域を読み込んでダンプ・リストで確認した例です．

予約領域はブート・セクタで，システムのイニシャル・プログラム・ローダ(Initial Program Loader：IPL)が入っています．

FAT(File Allocation Table)領域とは，ファイルやディレクトリを構成しているクラスタのリンク状態や使用クラスタ，不良クラスタなどに関する情報を集めたテーブルのことで，これは非常に重要な役割をもっています．そして，もしFATに欠陥が生じても，スペアのFATによりディスクの修復が可能なように通常二つのまったく同一のFATが作られます．

ルート・ディレクトリ領域には，ディレクトリ・エントリが収められています．ディレクトリ・エントリとは，ファイルの名前やディレクトリ名，ファイルの属性やFATのエントリ番号など，ファイルに付随するいろいろな情報の入ったテーブルです．

そして，データ領域に実際のファイルの内容が入っています．

これらの各領域の大きさは，**表4-1** に示したようにディスクの種類によって異なります．したがって，デバッガ symdeb などで直接各領域やファイルの内容などを見たい場合の論理的なセクタ番号は，次の式によって同表を参照して計算することができます．

　データ領域の開始セクタ
　＝予約セクタ数＋FATセクタ数
　　＋ルート・ディレクトリ サイズ
　ただし，
　FATセクタ数＝FAT数×(1FAT当たりのセクタ数)
　ルート・ディレクトリ・サイズ
　＝ディレクトリ数×32バイト/セクタ・サイズ

〔リスト4-1〕
ディスク上の領域

```
R>symdeb ⏎ …デバッガの起動
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-l 0 0 0 1⏎ } ブート・セクタのロードとダンプ
-d 0⏎
53C2:0000  EB 1C 90 4E 45 43 20 32-2E 30 30 00 04 01 01 00  k..NEC 2.00.....
53C2:0010  02 C0 00 D0 04 FE 02 00-08 00 02 00 00 00 33 C0  .@.P.~........3@
53C2:0020  8E D8 8E C0 8E D0 BC 8A-02 FC BE 0B 00 2E AD 3D  .X.@.P<..|>...-=
53C2:0030  80 00 75 38 BE B9 01 B8-00 0A CD 18 B4 0C CD 18  ..u8>9.8..M.4.M.
53C2:0040  B4 12 CD 18 0E 1F 33 C0-8E C0 B8 00 A0 26 F6 06  4.M...3@.@8. &v.
53C2:0050  01 05 08 74 03 B8 00 E0-8E C0 BF 40 01 AC 0A C0  ...t.8. .@?@.,.@
53C2:0060  74 04 AA 47 EB F7 B0 06-E6 37 EB FE 3D 00 02 75  t.*Gkw0.f7k~=..u
53C2:0070  0E 2E 83 7C 0D 01 74 BC-2E 80 7C 08 FD 74 B5 A0  ...|..t<..|.}t5
-l 0 0 1 1⏎ } FATセクタのロードとダンプ
-d 0⏎
53C2:0000  FE FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B  ~...@.. ..... ..
53C2:0010  C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60  @.. ..... ..@..`
53C2:0020  01 17 80 01 19 A0 01 1B-C0 01 1D E0 01 1F 00 02  ..... ..@.. ....
53C2:0030  21 20 02 23 40 02 25 60-02 27 80 02 29 A0 02 2B  ! .#@.%`.'..) .+
53C2:0040  C0 02 2D E0 02 2F 00 03-31 20 03 33 40 03 35 60  @.-`./..1 .3@.5`
53C2:0050  03 37 80 03 39 A0 03 3B-C0 03 3D E0 03 3F 00 04  .7..9 .;@.=`.?..
53C2:0060  41 F0 FF 43 40 04 45 60-04 47 80 04 49 A0 04 4B  Ap.C@.E`.G..I .K
53C2:0070  C0 04 4D 20 05 FF FF FF-FF FF FF 53 40 05 55 60  @.M .......S@.U`
-l 0 0 3 1⏎ } 予備のFATセクタのロードとダンプ
-d 0⏎
53C2:0000  FE FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B   ...@.. ..... ..
53C2:0010  C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60  @.. ..... ..@..`
53C2:0020  01 17 80 01 19 A0 01 1B-C0 01 1D E0 01 1F 00 02  ..... ..@.. ....
53C2:0030  21 20 02 23 40 02 25 60-02 27 80 02 29 A0 02 2B  ! .#@.%`.'..) .+
53C2:0040  C0 02 2D E0 02 2F 00 03-31 20 03 33 40 03 35 60  @.-`./..1 .3@.5`
53C2:0050  03 37 80 03 39 A0 03 3B-C0 03 3D E0 03 3F 00 04  .7..9 .;@.=`.?..
53C2:0060  41 F0 FF 43 40 04 45 60-04 47 80 04 49 A0 04 4B  Ap.C@.E`.G..I .K
53C2:0070  C0 04 4D 20 05 FF FF FF-FF FF FF 53 40 05 55 60  @.M .......S@.U`
-l 0 0 5 1⏎ } ルート・ディレクトリのロードとダンプ
-d 0⏎
53C2:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....
53C2:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  .........m.......
53C2:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....
53C2:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  .........m.B.@r..
53C2:0040  42 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  B           ....
53C2:0050  00 00 00 00 00 00 45 5F-48 12 4E 00 00 00 00 00  ......E_H.N.....
53C2:0060  41 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  A           ....
53C2:0070  00 00 00 00 00 00 46 5F-48 12 4F 00 00 00 00 00  ......F_H.O.....
-q⏎

R>
```

例として，1Mバイト・フォーマット・ディスクと640Kバイト・フォーマット・ディスクの各領域の開始セクタを**表4-1**から算出してみましょう．

● 1Mバイト・フォーマットの場合

予約セクタ=1
FAT数=2
1FAT当たりのセクタ数=2
ディレクトリの数=192
セクタ・サイズ=1024

これらの条件から，

FATセクタ=2×2=4
ルート・ディレクトリ・サイズ
=192×32/1024=6

よって，1Mバイト・フォーマットの場合に各領域の配置は**図4-2**(a)のようになります．

● 640Kバイト・フォーマットの場合

予約セクタ=1
FAT数=2
1FAT当たりのセクタ数=2
ディレクトリの数=112
セクタ・サイズ=512

これらの条件から，

FATセクタ=2×2=4
ルート・ディレクトリ・サイズ=112×32/512=7

よって，640Kバイト・フォーマットの場合に各領域の配置は同図(b)のようになります．

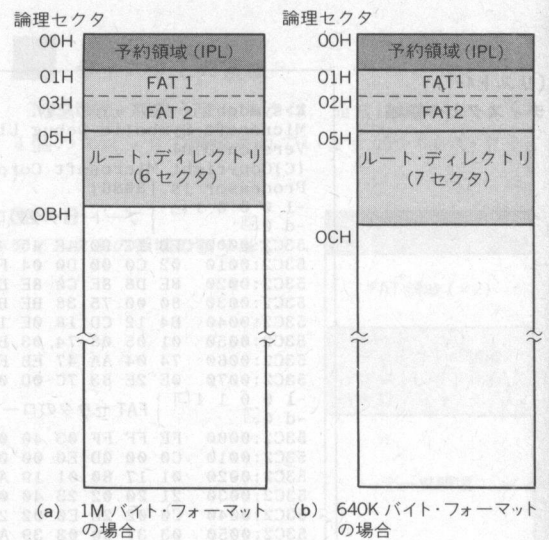
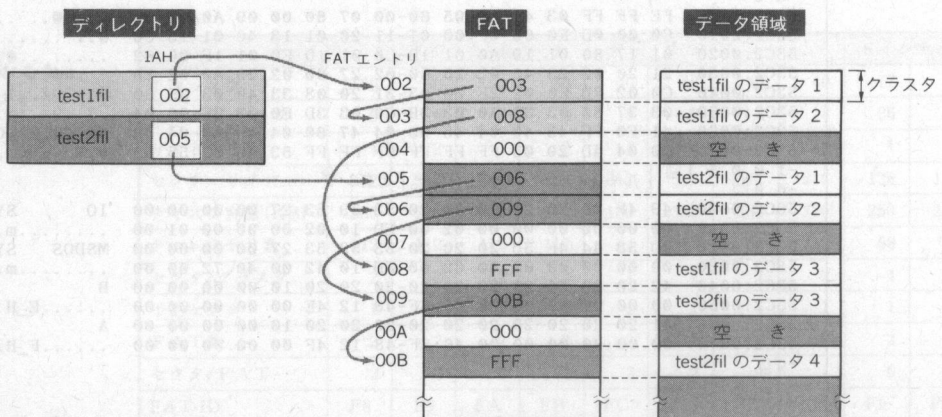
FATとクラスタ

ディレクトリやFATと，ディスク上のクラスタとの論理的な位置関係は**図4-3**のようになっていて，各クラスタをFATによりチェイン接続して，見かけ上ファイルが連続したクラスタに格納されているように管理されています．

詳しくは後述しますが，ここではディレクトリとFATおよびデータ領域のおおざっぱな関係について説明しておきましょう．MS-DOS ver.2.11では，FATテーブルのエントリが12ビット長で管理されていましたが，ver.3.10以降になって拡張された16ビット長のFATも使用できるようになりました．MS-DOSのFATでは，12ビット長のものが基本となるため，ここでは，FATエントリを12ビット長として解説します．

同図において，ファイルtest1filがアクセスされると，まずディレクトリ領域の中のファイル名の一致するディレクトリ・エントリ(各ファイルの管理情報の単

〔図4-2〕各領域の配置例

(a) 1Mバイト・フォーマットの場合　(b) 640Kバイト・フォーマットの場合

〔図4-3〕FATテーブルとデータ領域の関係

位：後述)が検索されます．そして，そのディレクトリ・エントリ(32バイト)の中のオフセット1AHには，FATエントリ(入口番号：002H)が書かれてあります．FATとデータ領域のクラスタは1対1に対応しているので，このFATエントリ002Hに対応するクラスタがtest1filの1番目のファイル内容になります．

次に，FATエントリの002Hには次のエントリ番号003Hが書かれているので，このFATエントリの003Hに対応するクラスタにtest1filの2番目のデータが格納されていることになります．

そして，FATエントリの003Hには次のエントリ番号008Hが書かれているので，その008Hに対応するクラスタがtest1filの3番目のデータ・ブロックということになります．

このようにクラスタのチェインが飛び飛びになるのは，サイズの異なるファイルを消去したあとに別のファイルを書き込んだ場合などによく起こり得ます．この場合に，空いているFATエントリには000Hが入り，対応するクラスタが空きクラスタであることを示しています．

そして，FATエントリの008HにはFFFHが入っているので，ここがtest1filファイルの最終クラスタであることがわかります．同様にtest2filは005H, 006H, 009H, 00BHとチェインされ，各々に対応するクラスタのデータをつないだものがtest2filの内容ということになります．

ここで注意したいことは，このように何度かファイルの削除や書き込みが繰り返されると，ファイルが飛び飛びにチェインされていることが予想され，それにともなってディスク・ドライブのヘッドが頻繁にシークされなければならず，読み出しや書き込みの処理速度が低下してしまうことです．

diskcopyコマンドでは，ディスク単位でまったく同じ内容のコピーを行うので，これらのFATの内容もそのままコピーされるためクラスタの整理は行われません．これに対し，新たにフォーマットされたディスクにcopyコマンドでファイル・コピーを行うと，copyコマンドではファイル単位でコピーを行うので，これらのFATのチェインは連続的に整理され，新しいディスクでのファイルのアクセス速度が向上することになります．

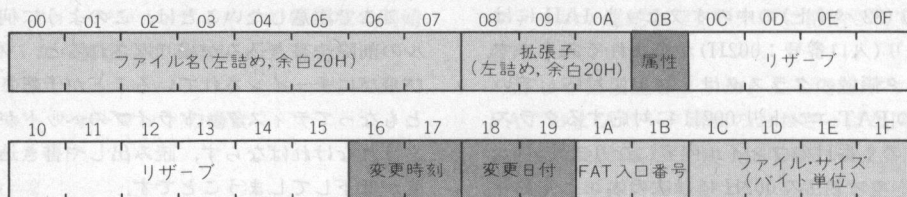
ディレクトリ・エントリ

ルート・ディレクトリのダンプ・リストの例をリスト4-2に示します．ディレクトリには，ディスク・ファイルの情報だけでなく，サブ・ディレクトリの情報やボリューム・ラベルの情報なども記録されています．

同リストのように，ディレクトリの中にはファイル名やサブ・ディレクトリ名のほかに，それに付随する

〔リスト4-2〕
ルート・ディレクトリの構成

```
R>symdeb ↵ …デバッガの起動
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086] ルート・ディレクトリのロードとダンプ
-l 0 0 5 1 ↵                                      FATエントリ
-d 0 1200  ↵  ファイルのベース名        拡張子    属性       サイズ
53C2:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....
53C2:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......
53C2:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....
53C2:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..
                                  時刻  日付
53C2:0040  42 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  B          .....
53C2:0050  00 00 00 00 00 00 45 5F-48 12 4E 00 00 00 00 00  ......E_H.N.....
53C2:0060  41 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  A          .....
53C2:0070  00 00 00 00 00 00 46 5F-48 12 4F 00 00 00 00 00  ......F_H.O.....
53C2:0080  46 49 4C 45 43 20 20 20-20 20 20 20 00 00 00 00  FILEC      .....
53C2:0090  00 00 00 00 00 00 6F 5F-48 12 50 00 11 00 00 00  ......o_H.P.....
53C2:00A0  43 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  C          .....
53C2:00B0  00 00 00 00 00 00 10 60-48 12 6A 00 00 00 00 00  .......`H.j.....
53C2:00C0  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....
                                サブ・ディレクトリ
53C2:00D0  00 00 00 00 00 00 02 00-ED 10 6B 00 63 61 00 00  ........m.k.ca..
53C2:00E0  E5 41 20 20 20 20 20 20-20 20 20 20 00 00 00 00  eA         .....
53C2:00F0  00 00 00 00 00 00 5C 5F-48 12 86 00 11 00 00 00  ......¥_H.......
53C2:0100  54 45 53 54 2D 44 49 53-4B 20 20 28 00 00 00 00  TEST-DISK  (....
53C2:0110  00 00 00 00 00 00 6D 60-48 12 00 00 00 00 00 00  ......m`H.......
53C2:0120  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  .eeeeeeeeeeeeeee
                              ボリューム・ラベル
53C2:0130  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  eeeeeeeeeeeeeeee
53C2:0140  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  .eeeeeeeeeeeeeee
53C2:0150  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  eeeeeeeeeeeeeeee
           未使用
           消去されたファイル

-q ↵

R>
```

〔図4-4〕ディレクトリ・エントリのフォーマット

いろいろな情報が記録されていますが，これらの情報のまとまりをディレクトリ・エントリと呼び，32バイトで構成されています．

● **ディレクトリ・エントリの構造**

ディレクトリ・エントリのフォーマットは**図4-4**のようになっています．以下，このディレクトリ・エントリのフォーマットについて解説していくことにします．

◆ **ファイルのベース名および拡張子(00H〜0AH)**

ファイル名と拡張子の入るフィールドで，その名前が左詰めで入ります．ここで，ファイル名が漢字の場合にはシフトJISコードが入ります．また余白には空白コード(20H)が入ります．

一度登録したファイルが削除されると，ファイル名の先頭がE5Hに書き替えられ，このディレクトリ・エントリが使用可能であることを表しています．

また，ディスクがフォーマットされたまま一度も使われたことのないエントリには，その先頭のバイトに00Hが入っていて，そのディレクトリ・エントリが未使用なことを表しています．

◆ **ファイルの属性(0BH)**

ディレクトリ・エントリの0BHには，ファイルの属性を表す値が入っていて，それぞれのビットの意味は**表4-2**に示すようになっています．また，これらのビットはそれぞれのビットが“1”のとき有効になります．

▶ **ビット0**

読み出し専用ファイルであることを表しています．このビットがセットされているファイルは，書き込みやdelコマンドによるファイルの削除ができません．

▶ **ビット1**

このビットはシークレット属性を表しています．このビットがセットされているファイルは“隠された”ファイルであることを表し，dirコマンドでは表示されません．

また，基本FCBを用いたシステム・コールで読み出そうとしてもエラーになります．この属性の付いたファイルをアクセスしたい場合には，拡張FCBを用いるか，UNIX流のファイル・ハンドルによるシステム・コールを使えばアクセスすることが可能です．

〔表4-2〕属性のビット

属性ビット								属性
×	×	×	×	×	×	×	1	読み出し専用ファイル
×	×	×	×	×	×	1	×	隠されたファイル
×	×	×	×	×	1	×	×	システム・ファイル
×	×	×	×	1	×	×	×	ボリューム・ラベル
×	×	×	1	×	×	×	×	サブ・ディレクトリ
×	×	1	×	×	×	×	×	通常のファイル

▶ **ビット2**

このビットがセットされているファイルはシステム・ファイルであることを表しています．io.sysやmsdos.sysなどのファイルにはこの属性がセットされています．

▶ **ビット3**

この属性のエントリは，ファイルに関する情報ではなく，ディスクのフォーマット時に指定したボリューム・ラベルであることを表しています．この場合，ファイル名と拡張子のフィールドにはボリューム名がセットされます．また，日付や時刻のフィールドには，そのディスクをフォーマットした時点の日付や時刻が記録されます．

▶ **ビット4**

この属性のついたエントリは，サブ・ディレクトリであることを表しています．

▶ **ビット5**

ファイルに対して書き込みが行われると，このビットが“1”にセットされます．通常のディスク・ファイルには，すべてこの属性がセットされることになります．

これらの属性は，それぞれの属性に矛盾がない限り自由に組み合わせて使用できます．

◆ **更新日時(16H〜17H，18H〜19H)**

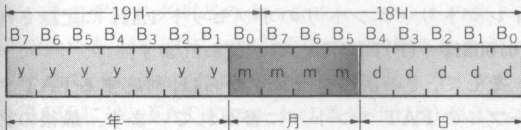
ディレクトリ・エントリの16H〜19Hのバイトには，そのファイルを更新(作成)した時刻(16H〜17H)と日付(18H〜19H)が記録されていて，その記録フォーマットは図4-5のようになっています．

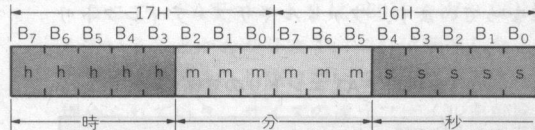
◆ **FATエントリ番号(1AH〜1BH)**

ディレクトリ・エントリの1AHバイトには，FAT

〔図4-5〕日付および時刻の記録フォーマット

〔日付の記録フォーマット〕

19H								18H							
B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
y	y	y	y	y	y	y	m	m	m	m	d	d	d	d	d
年							月				日				

年：1980を引いた値
月：1～12
日：1～31

〔時刻の記録フォーマット〕

時：0～23
分：0～59
秒：0～29（実際の秒の$\frac{1}{2}$）

のエントリ番号が入っています．これによりファイルの本体がディスク上のどこから始まっているかを知ることができます．FATに関しては，この後で詳しく解説してあります．

◆ **ファイル・サイズ（1CH～1FH）**

ディレクトリ・エントリの最後の4バイトには，そのファイルのサイズ（大きさ）をバイトで表現した値が入っています．このエントリがサブ・ディレクトリやボリューム名の場合は，このフィールドは00Hで満たされています．

FAT

FATとは，ファイルやディレクトリがディスク上のどのセクタに格納されているのか，またディスク・セクタの連結や空き領域および不良セクタなど，ディスク上のセクタの状況を記録している管理データ・テーブルのことをいいます．

● 12/16ビットのFAT長

前述のように，このFATで表現される単位にはセクタではなくクラスタという単位が使用されます．クラスタは，物理セクタの2のべき乗になるようにディスクの種類によって決まっています（**表4-1参照**）．MS-DOSでは，このFATの管理方法として，12ビットで管理するバージョンと16ビットで管理するバージョンとに分類されます．

ver.2.11までのバージョンでは，これらのクラスタの使用状況は000H～FFFHまでの12ビット（1.5バイト）の値で表現されていました．これに対し，ver.3.10以降では16ビット（2バイト）で表現するFATもサポートされるようになりました．

1983年にver.2.11がリリースされたころは，パーソナル・コンピュータのディスク・システムとしてフロッピ・ディスクが一般的であり，その容量も1Mバイト程度であったために12ビットFATエントリでも十分に機能していました．ところが，昨今になって周辺装置の価格も下がりハード・ディスクが一般的になってくると，その容量も20～100Mバイトとなってしまい，12ビットFATエントリでは支障をきたすようになってきました．すなわち，12ビットFATエントリではエントリ数（＝クラスタ数）が002H～FF6Hの4085個までしか扱えないため，たとえば20Mバイトのハード・ディスクでは1クラスタ当たり8Kバイトとなってしまいます．

これではディスクの記憶領域を有効に使うことはできません．なぜならMS-DOSでは，ディスクをクラスタ単位で管理しているため，たった1バイトのファイルでも1クラスタ（8Kバイト）を占有してしまい，残りのバイト数（ディスク容量）は無駄になってしまうからです．

この問題を解決するため，ver.3.10以降ではFATエントリの拡張を行い，従来の12ビットFATエントリに加えて16ビットFATエントリも扱えるようになりました．16ビットFATエントリでは0002H～FFF6Hまでの65525個のエントリが使用可能となり，1クラスタ当たり1Kバイトとしても，約60Mバイトのディスクまでサポートすることができます．

MS-DOSは，12ビットFATエントリと16ビットFATエントリの区別を，デバイス・ドライバから返されるBPB（BIOS Parameter Block：第8章）の各パラメータからクラスタ数を算出して自動的に判定しています*．

*：ここで苦言を一言．NECからリリースされているver.3.10では，せっかくMS-DOSが16ビットFATエントリをサポートしているにもかかわらず，io.sys内のハード・ディスク・ドライバやformatコマンドが16ビット対応となっていないため，拡張フォーマットを行うことができません（ハイ・レゾリューション・モードを除く）．そしてver.3.30になって，ようやくノーマル・モードでも16ビットFATエントリ対応となりましたが，ver.3.30で拡張フォーマットを行ったディスクは，ver.3.10からはアクセスできなくなってしまいます．

これがver.2.11とver.3.Xの相違によって区別されるのなら納得できますが，同じver.3.Xでディスクの互換性がなくなってしまうのですから最悪といえます．これは，OEMであるNECのver.3.10リリース時点における対応の遅れ（怠慢）であり，「殿様商売もいい加減にしてくれ」といいたくなります．

● **FAT の構造とファイルの関係**

FAT に用いられるエントリ番号の意味は**表4-3** のようになっています．ファイルやサブ・ディレクトリの最初のクラスタに対応するFAT テーブルの位置＝FAT の入口番号(FAT エントリ)は，前述のようにディレクトリ・エントリのオフセット1AH に記録され，(0)002H～(F)FF6H の値が使われます．

〔表4-3〕FAT エントリの意味

FAT エントリの値		値の示す意味
16 ビット	12 ビット	
0000H	000H	未使用クラスタ
0001H	001H	この値は使用されない
0002H ～ FFF6H	002H ～ FF6H	次にチェインされるべきFAT のエントリ番号
FFF7H	FF7H	不良クラスタ
FFF8H ～ FFFFH	FF8H ～ FFFH	ファイルの最後のクラスタ

そして，その次のクラスタからは，対応するFAT テーブルのFAT エントリに書かれています．最後のクラスタに対応するFAT エントリには，(F)FF8H～(F)FFFH の値が入り，ファイルの最終クラスタであることを表します．この値は，一般には(F)FFFH が使われています．

リスト4-3 に示すディレクトリ・エントリのFAT エントリ番号およびFAT テーブルにしたがって，実際にファイルの内容を追ってみましょう．

まず，FAT テーブルの第1バイトには，FAT-ID と呼ばれる1バイトのデータが書かれてあります．これ

〔リスト4-3〕ディレクトリ・エントリとFAT

```
R>symdeb⏎ …デバッガの起動
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-l 0 0 5 1⏎  } ルート・ディレクトリのロードとダンプ
-d 0 1200⏎   }
53C2:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00   IO      SYS'....
53C2:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00   .........m......
53C2:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00   MSDOS   SYS'....
53C2:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00   .........m.B.@r..
53C2:0040  42 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00   B          .....
53C2:0050  00 00 00 00 00 00 45 5F-48 12 4E 00 00 00 00 00   ......E_H.N.....
53C2:0060  41 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00   A          .....
53C2:0070  00 00 00 00 00 00 46 5F-48 12 4F 00 00 00 00 00   ......F_H.O.....
53C2:0080  46 49 4C 45 43 20 20 20-20 20 20 20 00 00 00 00   FILEC       ....
53C2:0090  00 00 00 00 00 00 6F 5F-48 12 50 00 11 00 00 00   ......o_H.P.....
53C2:00A0  43 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00   C          .....
53C2:00B0  00 00 00 00 00 00 10 60-48 12 6A 00 00 00 00 00   .......`H.j.....
53C2:00C0  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00   COMMAND COM ....
53C2:00D0  00 00 00 00 00 00 02 00-ED 10 68 00 63 61 00 00   .........m.h.ca..
53C2:00E0  E5 58 45 20 20 20 20 20-41 53 4D 20 00 00 00 00   eXE     ASM ....
53C2:00F0  00 00 00 00 00 00 12 58-48 12 64 00 C8 05 00 00   .......XH.d.H...
53C2:0100  54 45 53 54 2D 44 49 53-4B 20 20 28 00 00 00 00   TEST-DISK  (....
53C2:0110  00 00 00 00 00 00 6D 60-48 12 00 00 00 00 00 00   ......m`H.......
53C2:0120  43 4F 4D 20 20 20 20 20-41 53 4D 20 00 00 00 00   COM     ASM ....
53C2:0130  00 00 00 00 00 00 27 58-48 12 66 00 32 06 00 00   ......'XH.f.2...
53C2:0140  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5   .eeeeeeeeeeeeeee

              FAT テーブルのロードとダンプ
-l 0 0 1 2⏎  }                 IO.SYS
-d 0 1100⏎   }
53C2:0000  FE FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B   ...@.........
53C2:0010  C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60   @..........@..
53C2:0020  01 17 80 01 19 A0 01 1B-C0 01 1D E0 01 1F 00 02   .........@.....
53C2:0030  21 20 02 23 40 02 25 60-02 27 80 02 29 A0 02 2B   ! .#@.%`.'..) .+
53C2:0040  C0 02 2D E0 02 2F 00 03-31 20 03 33 40 03 35 60   @.-`./..1 .3@.5`
53C2:0050  03 37 80 03 39 A0 03 3B-C0 03 3D E0 03 3F 00 04   .7..9 .;@.=`.?..
53C2:0060  41 F0 FF 43 40 04 45 60-04 47 80 04 49 A0 04 4B   Ap.C@.E`.G..I .K
53C2:0070  C0 04 4D 20 05 FF FF FF-FF FF FF 53 40 05 55 60   @.M .......S@.U`
53C2:0080  05 57 80 05 59 A0 05 5B-C0 05 5D E0 05 5F 00 06   .W..Y .[@.]`._..
53C2:0090  61 20 06 FF 0F 00 00 00-00 67 F0 FF 69 B0 06 FF   a ......gp.i0.
53C2:00A0  CF 06 6D E0 06 6F 00 07-71 20 07 73 40 07 75 60   O.m`.o..q .s@.u`
53C2:00B0  07 77 80 07 79 A0 07 7B-C0 07 7D E0 07 7F 00 08   .w..y .{@.}`....
53C2:00C0  81 F0 FF 00 00 00 FF FF-FF 00 F0 FF FF 0F 00 00   .p.......p.....
53C2:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
53C2:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
53C2:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-q⏎
R>
```

FAT-ID（53C2:0000 の FE）／msdos.sys（53C2:0060 の 43 40）／command.com（53C2:0090 の 69 B0 06 FF）／command.com の終わり（53C2:00C0 の F0 FF）／ここで飛ぶ／ファイル終わり

は，表4-1に示したMS-DOS標準ディスク・フォーマットの場合に使用され，このFAT-IDバイトの値により，そのディスクがどのようなフォーマットのディスクなのかをMS-DOSが判別するのに利用されます．

(1) さて，io.sysのディレクトリ・エントリのオフセット1AHには，FATのエントリ番号(002H)が入っています．このFATエントリとFATテーブル内の開始番地は，一つのデータが12ビット(1.5バイト)で表現されているので，3バイト単位で1かたまりのデータとみなし，次の式で求めることができます．

FAT開始番地＝(エントリ番号÷2)×3

よって，このio.sysの場合は，

FAT開始番地＝(002H÷2)×3＝3

となり，FATテーブルの3バイト目にあることがわかります．FATテーブルのデータから，その次のFATエントリ番号は以下の手順から求めることができます．

① FATの開始番地からの3バイト(24ビット)のデータを抽出し，1バイトずつ逆順に並べてこれを12ビット(1.5バイト)ずつに分ける．

03 40 00 → 00 40 03 → 004 003

② 12ビット(1.5バイト)ずつのデータを入れ替える．

004 003 → 003 004

③ FATエントリ番号が偶数の場合は，この入れ替えたデータの最初の12ビット(1.5バイト)が次のFATエントリ番号になる．一方，FATエントリ番号が奇数の場合は，この入れ替えたデータの後のデータが次のFATエントリ番号になる．

したがって，002Hエントリの内容は003Hエントリへ，その次が004Hエントリへと続いていることがわかる．

(2) 次に，msdos.sysの場合はどうでしょうか．FATエントリが042Hなので上の式からFAT開始番地を求めます．

FAT開始番地＝(042H÷2)×3
＝21H×3
＝63H

そして，このFAT開始番地から次のFATエントリを求めます．

43 40 04 → 04 40 43 → 044 043 → 043 044

となり，043H，044Hと続いていることがわかります．

(3) ほかのファイルも同様にして，FATエントリを簡単に求めることができます．ここではcommand.comについて調べてみましょう．

FAT開始番地＝(68H÷2)×3
＝34H×3
＝9CH

FATテーブルのオフセット9CHからの3バイトずつを抽出して分解します．

69 B0 06 → 06 B0 69 → 06B 069 → 069 06B

ここで，FATエントリは069Hの次に06BHに飛んでいることがわかります．これは前述したように，ファイルを書き込む際に空いているクラスタから順次埋められていき，サイズの違うファイルを削除したり書き込んだりしているうちに飛び飛びのクラスタに書かれるためにこのようなことが起こります．

この06BHから，改めて次のFAT開始番地を求めます．

FAT開始番地＝(06BH÷2)×3
＝35H×3
＝9FH

FATテーブルの9FHから，

FF CF 06 → 06 CF FF → 06C FFF → FFF 06C
6D E0 06 → 06 E0 6D → 06E 06D → 06D 06E
6F 00 07 → 07 00 6F → 070 06F → 06F 070
71 20 07 → 07 20 71 → 072 071 → 071 072
73 40 07 → 07 40 73 → 074 073 → 073 074
75 60 07 → 07 60 75 → 076 075 → 075 076
77 80 07 → 07 80 77 → 078 077 → 077 078
79 A0 07 → 07 A0 79 → 07A 079 → 079 07A
7B C0 07 → 07 C0 7B → 07C 07B → 07B 07C
7D E0 07 → 07 E0 7D → 07E 07D → 07D 07E
7F 00 08 → 08 00 7F → 080 07F → 07F 080
81 F0 FF → FF F0 81 → FFF 081 → 081 FFF

と続き，FATエントリがFFFHになったところで終了です．

これらのルート・ディレクトリとFAT，およびクラスタの関係を図で示したのが図4-6です．

階層ディレクトリの実現

さて，それでは階層ディレクトリの場合は，このFATやサブ・ディレクトリの管理方法はどのようになっているのでしょうか．

ここでは，ファイルやサブ・ディレクトリの関係が図4-7のようになっている場合を例にとって，サブ・ディレクトリやFATの関係を調べてみることにしましょう．リスト4-4はその実行例を表しています．

① まず，dirコマンドによりルート・ディレクトリを調べておく．

② 次に，デバッガsymdebを使ってルート・ディレクトリ領域の内容をダンプ・リストで確認する．

③ また，FATテーブルの内容もダンプ・リストで確認する．そして，まずディレクトリ¥Bの内容を調べてみる．

〔図4-6〕 実例における FAT とクラスタのチェイン

ルート・ディレクトリ

ファイル	開始クラスタ
io.sys	002
msdos.sys	042
b¥	04E
a¥	04F
filec	050
c¥	06A
command.com	068
com.asm	066

FAT テーブル／クラスタ

FAT 番号	FAT 内容	クラスタ	内容
002	003	00B	io.sys の内容（連続している）
003	004	00C	
⋮		⋮	
040	041	049	
041	FFF	04A	
042	02B	04B	msdos.sys の内容（連続している）
043	02C	04C	
⋮		⋮	
04C	03B	055	
04D	FFF	056	
04E	FFF	057	サブ・ディレクトリb¥
04F	FFF	058	サブ・ディレクトリa¥
050	FFF	059	filec
⋮		⋮	
066	FFF	06F	com.asm
067	000	070	
068	069	071	command.com
069	06B	072	
06A	FFF	073	サブ・ディレクトリc¥
06B	06C	074	command.com のつづき
06C	06D	075	
⋮		⋮	
081	FFF	08A	

〔図4-7〕 例題の階層ディレクトリ

〔リスト4-4〕階層ディレクトリの実現 ①

```
R>dir a:¥ ↵…ルート・ディレクトリの確認①

 ドライブ A: のディスクのボリュームラベルは TEST-DISK
 ディレクトリは A:¥

B            <DIR>      89-02-08   11:58
A            <DIR>      89-02-08   11:58
FILEC               17  89-02-08   11:59
C            <DIR>      89-02-08   12:00
COMMAND  COM     24931  88-07-13    0:00
COM      ASM      1586  89-02-08   11:01
        6 個のファイルがあります.
  1118208 バイトが使用可能です.

R>symdeb ↵…デバッガの起動
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-l 0 0 5 1↵  }
-d 0 1200↵   } ルート・ディレクトリ領域のロードとダンプ②
53C2:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....
53C2:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......
53C2:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....
53C2:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..
53C2:0040  42 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  B          .....
53C2:0050  00 00 00 00 00 00 45 5F-48 12 4E 00 00 00 00 00  ......E_H.N.....
53C2:0060  41 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  A          .....
53C2:0070  00 00 00 00 00 00 46 5F-48 12 4F 00 00 00 00 00  ......F_H.O.....
53C2:0080  46 49 4C 45 43 20 20 20-20 20 20 20 00 00 00 00  FILEC       ....
53C2:0090  00 00 00 00 00 00 6F 5F-48 12 50 00 11 00 00 00  ......o_H.P.....
53C2:00A0  43 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  C          .....
53C2:00B0  00 00 00 00 00 00 10 60-48 12 6A 00 00 00 00 00  .......`H.j.....
53C2:00C0  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....
53C2:00D0  00 00 00 00 00 00 02 00-ED 10 68 00 63 61 00 00  ........m.h.ca..
53C2:00E0  E5 58 45 20 20 20 20 20-41 53 4D 20 00 00 00 00  eXE     ASM ....
53C2:00F0  00 00 00 00 00 00 12 58-48 12 64 00 C8 05 00 00  .......XH.d.H...
53C2:0100  54 45 53 54 2D 44 49 53-4B 20 20 28 00 00 00 00  TEST-DISK  (....
53C2:0110  00 00 00 00 00 00 6D 60-48 12 00 00 00 00 00 00  ......m`H.......
53C2:0120  43 4F 4D 20 20 20 20 20-41 53 4D 20 00 00 00 00  COM     ASM ....
53C2:0130  00 00 00 00 00 00 27 58-48 12 66 00 32 06 00 00  ......'XH.f.2...
53C2:0140  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  .eeeeeeeeeeeeeee

-l 0 0 1 1↵  }
-d 0 1100↵   } FATテーブルのロードとダンプ③
53C2:0000  FE FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B  ~...@..`........
53C2:0010  C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60  @.......  ..@..`
53C2:0020  01 17 80 01 19 A0 01 1B-C0 01 1D E0 01 1F 00 02  ..... ..@.......
53C2:0030  21 20 02 23 40 02 25 60-02 27 80 02 29 A0 02 2B  ! .#@.%`.'..) .+
53C2:0040  C0 02 2D E0 02 2F 00 03-31 20 03 33 40 03 35 60  @.-../..1 .3@.5`
53C2:0050  03 37 80 03 39 A0 03 3B-C0 03 3D E0 03 3F 00 04  .7..9 .;@.=`.?..
53C2:0060  41 F0 FF 43 40 04 45 60-04 47 80 04 49 A0 04 4B  Ap.C@.E`.G..I .K
53C2:0070  C0 04 4D 20 05 FF FF FF-FF FF FF 53 40 05 55 60  @.M .......S@.U`
53C2:0080  05 57 80 05 59 A0 05 5B-C0 05 5D E0 05 5F 00 06  .W..Y .[@.]`._..
53C2:0090  61 20 06 FF 0F 00 00 00-00 67 F0 FF 69 B0 06 FF  a .......gp.i0..
53C2:00A0  CF 06 6D E0 06 6F 00 07-71 20 07 73 40 07 75 60  O.m`.o..q .s@.u`
53C2:00B0  07 77 80 07 79 A0 07 7B-C0 07 7D E0 07 7F 00 08  .w..y .{@.}`....
53C2:00C0  81 F0 FF 00 00 00 FF FF-FF 00 F0 FF FF 0F 00 00  .p.........p....
53C2:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
53C2:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
53C2:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
-q↵

R>dir a:¥b ↵…ディレクトリ内容の確認④

 ドライブ A: のディスクのボリュームラベルは TEST-DISK
 ディレクトリは A:¥B

.            <DIR>      89-02-08   11:58
..           <DIR>      89-02-08   11:58
FILEB               17  89-02-08   11:59
        3 個のファイルがあります.
  1118208 バイトが使用可能です.

R>symdeb ↵…デバッガの起動
Microsoft Symbolic Debug Utility
```

〔リスト4-4〕階層ディレクトリの実現 ②

```
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-l 0 0 4e+9 1 ⏎ } ディレクトリ(¥B) 内容のロードとダンプ ⑤
-d 0 1100 ⏎
53C2:0000  2E 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  .          .....
53C2:0010  00 00 00 00 00 00 45 5F-48 12 4E 00 00 00 00 00  ......E_H.N.....
53C2:0020  2E 2E 20 20 20 20 20 20-20 20 20 10 00 00 00 00  ..         .....
53C2:0030  00 00 00 00 00 00 45 5F-48 12 00 00 00 00 00 00  ......E_H.......
53C2:0040  46 49 4C 45 42 20 20 20-20 20 20 20 00 00 00 00  FILEB       ....
53C2:0050  00 00 00 00 00 00 6A 5F-48 12 87 00 11 00 00 00  ......j_H.......
53C2:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
                              .
                              .
                              .
-l 0 0 87+9 1 ⏎ } ファイル (FILEB) 内容のロードとダンプ
-d 0 1100 ⏎
53C2:0000  74 69 68 73 20 69 73 20-66 69 6C 65 20 42 0D 0A  tihs is file B..
53C2:0010  1A 00 00 00 00 00 00 00-00 00 00 00 00 00 02 02  ................
                              .
                              .
                              .
-q⏎

R>dir a:¥a ⏎ … ディレクトリ内容の確認 ⑥

 ドライブ A: のディスクのボリュームラベルは TEST-DISK
 ディレクトリは  A:¥A

.            <DIR>      89-02-08   11:58
..           <DIR>      89-02-08   11:58
AA           <DIR>      89-02-08   12:01
FILEA                17 89-02-08   11:58
        4 個のファイルがあります.
  1118208 バイトが使用可能です.

R>dir a:a¥aa ⏎ … ディレクトリ内容の確認

 ドライブ A: のディスクのボリュームラベルは TEST-DISK
 ディレクトリは  A:¥A¥AA

.            <DIR>      89-02-08   12:01
..           <DIR>      89-02-08   12:01
FILEA                17 89-02-08   11:58
        3 個のファイルがあります.
  1118208 バイトが使用可能です.

R>symdeb ⏎ … デバッガの起動
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-l 0 0 4f+9 1 ⏎ } ディレクトリ(¥A) 内容のロードとダンプ ⑦
-d0 1100 ⏎
53C2:0000  2E 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  .          .....
53C2:0010  00 00 00 00 00 00 46 5F-48 12 4F 00 00 00 00 00  ......F_H.O.....
53C2:0020  2E 2E 20 20 20 20 20 20-20 20 20 10 00 00 00 00  ..         .....
53C2:0030  00 00 00 00 00 00 46 5F-48 12 00 00 00 00 00 00  ......F_H.......
53C2:0040  41 41 20 20 20 20 20 20-20 20 20 10 00 00 00 00  AA         .....
53C2:0050  00 00 00 00 00 00 20 60-48 12 84 00 00 00 00 00  ...... `H.......
53C2:0060  46 49 4C 45 41 20 20 20-20 20 20 20 00 00 00 00  FILEA       ....
53C2:0070  00 00 00 00 00 00 5C 5F-48 12 85 00 11 00 00 00  ......¥_H.......
53C2:0080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
                              .
                              .
                              .
```

ルート・ディレクトリのFATエントリ番号からFATテーブルのアドレスを計算する.

FAT開始番地＝(04EH MOD 2)×3

　　　　　　＝27H×3

　　　　　　＝75H

次に, FATテーブルのオフセット75Hからの3バイトを抽出して分解する.

FF FF FF → FF FF FF → FFF FFF → FFF FFF

ここでは, 前のほうのFFFHになる. したがって, ディレクトリ¥Bは1クラスタで構成されていて, こ

〔リスト4-4〕 階層ディレクトリの実現 ③

```
-l 0 0 85+9 1 ⏎ } ファイル (¥A¥FILEA) 内容のロードとダンプ ⑧
-d 0 1100 ⏎
53C2:0000  74 69 68 73 20 69 73 20-66 69 6C 65 20 41 0D 0A  tihs is file A..
53C2:0010  1A 30 20 2D 48 34 20 2D-49 20 2D 6E 20 2D 4D 31  .0 -H4 -I -n -M1
                              .
                              .
                              .
-l 0 0 84+9 1 ⏎ } ディレクトリ (¥A¥AA) 内容のロードとダンプ ⑨
-d 0 1100 ⏎
53C2:0000  2E 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  .           ....
53C2:0010  00 00 00 00 00 00 20 60-48 12 84 00 00 00 00 00  ...... `H.......
53C2:0020  2E 2E 20 20 20 20 20 20-20 20 20 10 00 00 00 00  ..          ....
53C2:0030  00 00 00 00 00 00 20 60-48 12 4F 00 00 00 00 00  ...... `H.O.....
53C2:0040  46 49 4C 45 41 20 20 20-20 20 20 20 00 00 00 00  FILEA       ....
53C2:0050  00 00 00 00 00 00 5C 5F-48 12 51 00 11 00 00 00  ......¥_H.Q.....
53C2:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
                              .
                              .
                              .
-l 0 0 51+9 1⏎ } ファイル (¥A¥AA¥FILEA) 内容のロードとダンプ ⑩
-d 0 1100⏎
53C2:0000  74 69 68 73 20 69 73 20-66 69 6C 65 20 41 0D 0A  tihs is file A..
53C2:0010  1A 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60  ...`..... ..@..`
                              .
                              .
                              .
-q⏎

R>dir a:¥c ⏎…ディレクトリ (¥C) の確認 ⑪

 ドライブ A: のディスクのボリュームラベルは  TEST-DISK
 ディレクトリは  A:¥C

.            <DIR>      89-02-08   12:00
..           <DIR>      89-02-08   12:00
FILEC                17  89-02-08   11:59
        3 個のファイルがあります.
    1118208 バイトが使用可能です.

R>symdeb ⏎…デバッガの起動
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-l 0 0 6a+9 1⏎ } ディレクトリ (¥C) 内容のロードとダンプ ⑫
-d 0 1100⏎
53C2:0000  2E 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  .           ....
53C2:0010  00 00 00 00 00 00 10 60-48 12 6A 00 00 00 00 00  .......`H.j.....
53C2:0020  2E 2E 20 20 20 20 20 20-20 20 20 10 00 00 00 00  ..          ....
53C2:0030  00 00 00 00 00 00 10 60-48 12 00 00 00 00 00 00  .......`H.......
53C2:0040  46 49 4C 45 43 20 20 20-20 20 20 20 00 00 00 00  FILEC       ....
53C2:0050  00 00 00 00 00 00 6F 5F-48 12 88 00 11 00 00 00  ......o_H.......
53C2:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
                              .
                              .
                              .
-l 0 0 88+9 1⏎ } ファイル (¥C¥FILEC) 内容のロードとダンプ ⑬
-d 0 1100⏎
53C2:0000  74 69 68 73 20 69 73 20-66 69 6C 65 20 43 0D 0A  tihs is file C..
53C2:0010  1A 00 0D 75 02 00 00 00-1E 00 00 00 01 00 04 00  ...u............
                              .
                              .
                              .
-q⏎

R>
```

の場合のディスク・フォーマットは1Mバイト・タイプなので，その内容の入っている論理レコード番号は4EH＋9＝57Hと求まる（9はFAT領域とディレクトリ領域）．

④ ここで，一度デバッガsymdebを終了し，dirコマンドを用いてディレクトリ¥Bの内容を確認しておく．

すると，“.”と“..”のサブ・ディレクトリが作られていることがわかる．これらはサブ・ディレクトリには必ず存在し，“.”はそのサブ・ディレクトリ自身を，また，“..”はその親ディレクトリを表している．

⑤ そして，あらかじめ求めておいた論理レコード番号を使用してディスクの内容を読み込み，ダンプ・リストを見ればディレクトリ¥Bの内容を見ることができる．

ディレクトリ¥Bのダンプ・リストから，そのサブ・ディレクトリ“.”のFATエントリ番号を調べると04EHになっていて，これはディレクトリ¥B自体のFATエントリ番号と一致する．したがって，ディレクトリ“.”はそのディレクトリ自体を表すことになる．

次に，ディレクトリ“..”の場合は，FATエントリ番号が000Hになっていて，ファイルの未使用を表すFATエントリ番号になっており，その親ディレクトリがルート・ディレクトリであることを表わしている(図4-7)．

⑥ 次に，このサブ・ディレクトリ¥Bの中にあるファイルを調べてみる．ファイルfilebのFATエントリ番号は087Hなので，デバッガsymdebのlコマンドおよびdコマンドを用いて，その内容の入っているレコードをダンプ・リストで確認する．

ここで，デバッガsymdebを終了し，dirコマンドを用いてディレクトリ¥Aを確認しておく．ディレクトリ¥Aには，ディレクトリAAとファイルfileaが入っている．また，ディレクトリ¥A¥AAには，やはりfileaが入っている．

⑦ 次に，ディレクトリ¥Aの内容を調べてみる．同リストのルート・ディレクトリのダンプ・リストから，ディレクトリ¥AのFATエントリは4FHなので，そのレコードの内容をダンプ・リストで確認する．

ここで，ディレクトリ“.”と“..”は，前述のディレクトリ¥Bと同様であることがわかる．

⑧ そして，この中にあるファイルfileaのFATエントリ番号(085H)を用いて，そのレコード内容をダンプ・リストで確認する．

⑨ では，このサブ・ディレクトリ¥Aのさらに下にあるサブ・ディレクトリAAの場合はどうなるか．

ディレクトリ¥A¥AAのFATエントリ番号(084H)を用いてディスク内容を読み出し，その内容をダンプ・リストで確認する．すると，ディレクトリの中にあるディレクトリ“.”は，前述のように自分自身(084H)を指しており，“..”のFATエントリ番号を見ると04FHを指している．

これは，ディレクトリ¥AのFATエントリであり，このディレクトリAAの親ディレクトリはディレクトリ¥Aであることがわかる．

⑩ また，このサブ・ディレクトリAAの中にあるファイルfileaは，FATエントリ051Hを用いてディスクから読み出し，メモリ・ダンプして確認することができる．

⑪ 次に，デバッガsymdebを終了し，dirコマンドを用いてディレクトリ¥Cの確認を行う．

⑫ そして，ルート・ディレクトリのダンプ・リストから，ディレクトリ¥CのFATエントリ06AHを知ることができるので，メモリ・ダンプによりディレクトリ¥Cの内容を確認する．ここで，ファイル¥C¥filecのFATエントリ088Hを知ることができる．

⑬ 次に，そのFATエントリ088Hを用いて，ディスク・ダンプを行ってファイル¥C¥filecの内容を確認できる．

このようにMS-DOSでは，階層ディレクトリの実現においても管理上は通常のファイルと大差はなく，FATによるクラスタのチェイン接続によって実現されています．

また，ここで注意したいことはルート・ディレクトリの場合は，そのエントリの大きさ(領域)がデバイス・ドライバのBPBテーブルによって規定され，ディレクトリ(ファイル)の数が固定になっているということです．これに対してサブ・ディレクトリの場合は，ディレクトリ・エントリの領域が通常のファイルと同様に，ディスク容量の範囲内でいくらでも可変できるため，ディレクトリ(ファイル)の数には制限がありません．

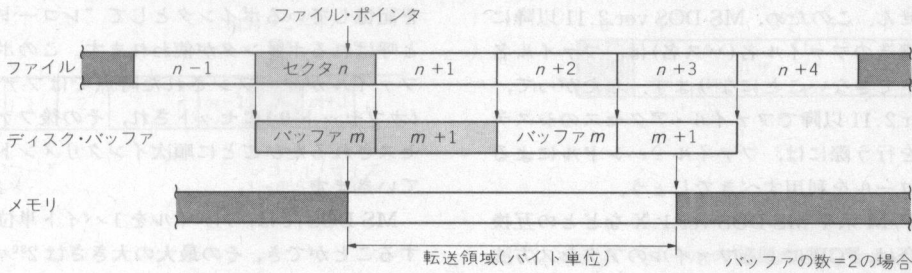
ディスク・バッファ

MS-DOSでは，ファイルに対するアクセスがバイト単位で行えるように設計されています．一方，ディスクの物理的な単位はセクタ・サイズ(ディスクの種類により異なる)によって決まっているので，このままではバイト単位によるアクセスはできないことになります．そこで，この問題を解決するためにディスクのバッファリングがあります．

図4-8は，ディスク・バッファリングの基本的な考えかたを示したものです．同図では，ディスク上のファイル・ポインタが指している位置から，メモリの転送領域へバイト単位で読み込もうとしています．すると，まずディスクからバッファへセクタ単位でファイルの内容が読み込まれます．

同図の例では，バッファの数が2個になっているためセクタnと$n+1$がバッファに転送され，その中からメモリの転送領域へ必要なバイト数が転送されます．次にバッファは空になっているのでセクタ$n+2$と$n+3$がやはり2セクタ分読み込まれます．そして，このバッファからメモリの転送領域へ必要なバイト数が読み込まれることになります．

このようにMS-DOSでは，ディスクとMS-DOSの

〔図4-8〕ディスクのバッファリング

間にバッファを置くことによって，ディスクからの読み書きはセクタ・サイズ(512，1024バイト/セクタなど)でバッファ(メモリ)との間で行い，MS-DOSで指定されたディスク・アクセスのデータ領域との転送はバイト転送を行うことにより，ファイルのバイト単位でのアクセスを可能にしています．

また，このディスク・バッファにはもう一つの重要な役割があります．それは，データの読み書きをいちいちディスクとの間で行うのではなくバッファとの間で行い，このバッファとディスクとの転送は必要最小限にとどめることにより，ディスク・アクセスの処理速度を向上させることにあります．

この場合，データだけでなく，ディレクトリ・エントリあるいはFATなどもこのバッファに読み込まれ保存されています．そして，同図のようにディスク・バッファがある限り次々にバッファが割り当てられていきます．

バッファが一杯になると次の順序にしたがってバッファの解放が行われます．

(1) 読み出し専用のバッファを解放する．
(2) 書き込み用バッファのデータをディスクに書き込んでバッファを解放する．
(3) FATやディレクトリのバッファをディスクに書き込んで解放する．

したがって，バッファの数(容量)が多いと，このバッファとの間でのデータ転送が多くなるため，ディスク・アクセスの処理速度も向上することになります．

しかし，これらのデータがバッファに残ったままディスクの交換が行われると，別のメディアにこれらのデータが書き込まれることになり，これはディスクの破壊につながります．そこで，これを解決するために，MS-DOSではディスクをアクセスするたびに，デバイス・ドライバに対してメディアのチェック・コマンドが発行されます．ここでデバイス・ドライバ側では，

(1) ディスクは交換されていない
(2) ディスクは交換された
(3) ディスクの交換は不明

の三つの中の一つの状態を示す数値をMS-DOSに返すことにより，これらの判定が行われるようになっていて，実際にはこれらの判定はディスク・ドライブのドア・オープンによって判定されます．

したがって，このディスク・バッファが有効に活用されているかどうかは，デバイス・ドライバ(io.sys)に依存することになります．このディスク・バッファが有効に活用されているかどうかは次の手順により判定できます．

(1) バッファの数を，たとえば10〜20個程度にセットする．
(2) システム・リセットをかける．
(3) ^Cを押してディスク・バッファをクリアする．
(4) dirコマンドを実行する(ここでディスク・アクセスが行われることを確認する)．
(5) もう一度dirコマンドを実行する．

この2回目のdirコマンドの実行時に再度ディスクのアクセスが行われると，そのシステムでは残念ながらディスクのバッファリングが有効に活用されていないことになります．2回目のdirコマンドの実行時にディスクのアクセスが行われないようであれば，ディスクのバッファが有効に活用されているシステムです．

この場合には，config.sysファイルのBUFFERSコマンドでバッファの数を適当に設定してやることにより，ディスク・アクセスにともなう処理速度が大幅に改善されることになります．

4-2 ファイル・ハンドルとFCB

MS-DOSでは，システム・コールを用いてファイルのアクセスを行う場合に，CP/M流のFCB(File Control Block)によるファイル・アクセスと，UINX流のファイル・ハンドルによるファイル・アクセスの2種類の方法をサポートしています．

このうち，FCBによるファイルのアクセスでは，フ

ァイル名と拡張子を含めて11文字までのファイル名しか扱えません．このため，MS-DOS ver.2.11以降における階層構造のファイル名(パス名)は，ファイル名としてセットできないことになります．したがって，MS-DOS ver.2.11以降でファイル・アクセスのシステム・コールを行う際には，ファイル・ハンドルによるシステム・コールを利用すべきでしょう．

しかし，CP/M系やMS-DOS ver.1.Xなどとの互換性を保つ場合は，FCBによるファイルのアクセス方法も理解しておかなければなりません．

■ FCB

FCBを用いたシステム・コールでファイルのアクセスを行う場合は，図4-9に示すフォーマットで，その内容が正しくセットされたメモリの先頭アドレスを，指定されたレジスタにセットしてMS-DOSに渡すことになっています．

ここで，FCBを用いたシステム・コールにおけるファイル・アクセス用のポインタについて述べておきましょう．

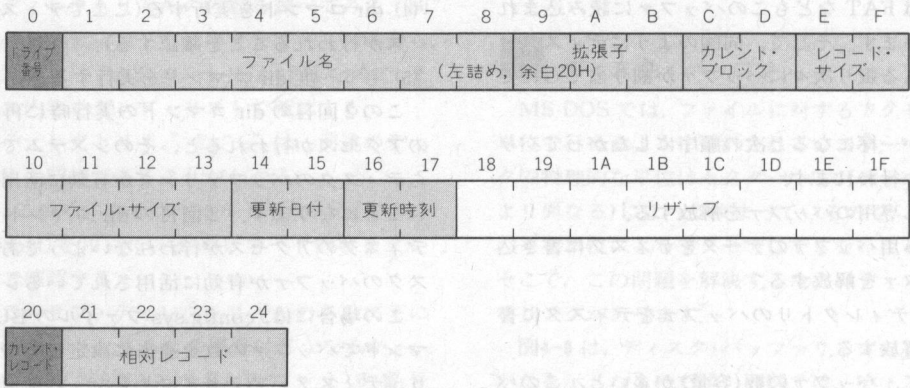
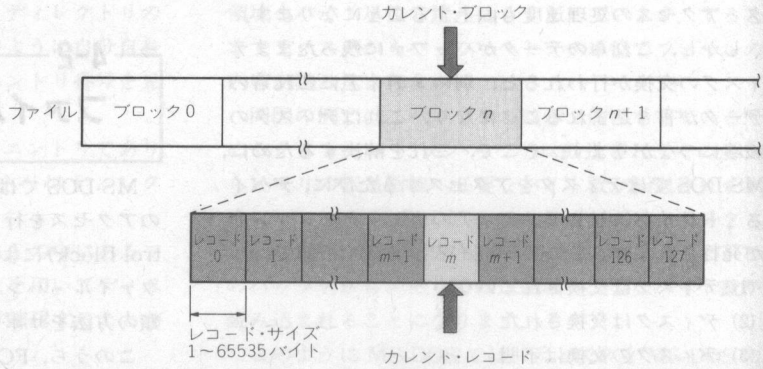
シーケンシャル・ファイルでは，ファイルの読み書きをしたのち，次回の読み書きをどこから行うべきかを記憶しているポインタとして“レコード・ポインタ”と呼ばれるポインタが使われます．このポインタは，ファイルがオープンされた時点ではファイルの先頭(オフセット0)にセットされ，その後ファイルがアクセスされるたびごとに順次インクリメント(増加)されていきます．

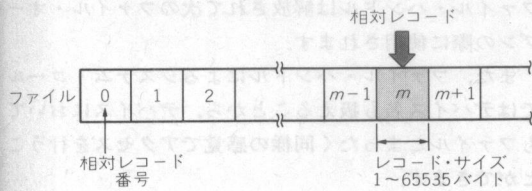
MS-DOSでは，ファイルを1バイト単位で読み書きすることができ，その最大の大きさは2^{30}バイト，すなわち1Gバイトまで扱えるようになっているので，レコード・ポインタとしては4バイトが必要となります．このシーケンシャル・ファイル用のレコード・ポインタは，FCB内のカレント・ブロック，カレント・レコード，レコード・サイズの三つのパラメータにより作られます(図4-10)．

またランダム・ファイルの場合は，そのポインタに相対レコードとレコード・サイズを使用することにより，目的のレコードにアクセスできようになっています(図4-11)．

● 基本FCBの構造

では，基本FCBの各フィールドについて解説して

〔図4-9〕基本FCBのフォーマット

〔図4-10〕
シーケンシャル・ファイルのアクセス

〔図4-11〕ランダム・ファイルのアクセス

〔図4-12〕拡張 FCB のフォーマット

いきましょう．

◆ **ドライブ番号(00H)**

オフセット 00H のドライブ番号は，00H でカレント・ドライブを指定します．以下 01H は A ドライブ，02H は B ドライブの順で指定できます．

00H を指定すると，ファイル・オープンのシステム・コールを行った際に，MS-DOS によってカレント・ドライブ番号に変換されてきます．

◆ **ファイルのベース名および拡張子(01H〜0BH)**

ファイル名は，8 文字までの ASCII コードを左詰めでセットし，余白には空白コード(20H)をセットします．同様に拡張子にも 3 文字までの ASCII コードでセットします．

ファイル名の検索や削除などのシステム・コールにおいては，このファイルのベース名や拡張子にワイルド・カード(*や？)を使用することができます．しかし，当然のことながらファイルのオープンやクローズ，あるいは読み出し/書き込みなどのシステム・コールではワイルド・カードは使用できません．

また，ファイル名にデバイス名をセットすることによって，デバイス・アクセスの場合でも通常のファイルと同様に扱うことができます．

◆ **カレント・ブロック(0CH〜0DH)**

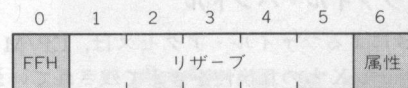
1 レコードを 128 個集めたデータのかたまりを“ブロック”と呼んでいます．

シーケンシャル・ファイルにおいて，ファイルの先頭から何ブロック目にレコード・ポインタが位置しているのかを表すものを“カレント・ブロック”と呼びます．

このフィールドには，ファイルをオープンした時点では“0”がセットされ，以後ファイルのアクセスが行われるたびにカレント・レコードとともにインクリメントされていきます(**図4-10**)．

◆ **レコード・サイズ(0EH〜0FH)**

レコード・サイズとは，読み出しまたは書き込みのシステム・コールにおいて，1 回に何バイト扱うかを指定するもので，MS-DOS では 1〜64 K バイトの範囲で可変できます．

このレコード・サイズを小さくすると，ユーザ・プログラム内でのバッファのサイズは小さくてすみますが，システム・コールの回数が多くなることになり，トータルでの処理速度は遅くなります．反面レコード・サイズを大きくするとまったく逆のことがいえます．

また，このフィールドに“0”をセットすると MS-DOS によってデフォルトの 128 がセットされます(**図4-10，4-11**)．

◆ **ファイル・サイズ(10H〜13H)**

ディレクトリ・エントリのオフセット 1CH〜1FH に記録されているファイルの大きさが 32 ビット整数で入ります．この最大値は 2^{30}(1 G)バイトです．

◆ **日付/時刻(14H〜17H)**

ファイルに対して，最後に書き込み(更新)を行った時点での日付と時刻がセットされ，ファイル・クローズのシステム・コールによってディレクトリ・エントリの 16H〜19H にこの内容が記録されます．

ここで，日付や時刻のフォーマットは前出の**図4-5** のようになっています．

◆ **カレント・レコード(20H)**

カレント・ブロック内の次にアクセスされるべきレコードの位置(0〜127)を示しています．この値は，ファイルのアクセスが行われたのち自動的にインクリメント(1 増加)されます．

そして，この値が 127 を越えると，今度はカレント・ブロックがインクリメントされてカレント・レコードは“0”にセットされます(**図4-10**)．

◆ **相対レコード(21H〜24H)**

相対レコードとは，ファイルをランダムにアクセスするシステム・コールで用いられ，アクセスすべきレコードが，ファイルの先頭から何番目のレコードにあるのかを指定するために使用されます(**図4-11**)．

● **拡張 FCB の構造**

拡張 FCB は，通常のファイル以外のファイル(シークレットやシステム・ファイルの属性の付いたファイル)をアクセスするときに使用される FCB です．拡張 FCB は，基本 FCB の前に**図4-12** に示すフォーマットの 7 バイトの拡張部を連続して配置することにより実現できます．また，属性については**表4-2**(112 ページ)にしたがって指定します．

ファイル・ハンドル

FCBによるファイル・アクセスは，CP/MやMS-DOS ver.1.Xとの互換性を考えて残されています．

しかし，実際にファイルのオープンや読み出し/書き込みのシステム・コールを行う際には，レコード・サイズやカレント・ブロック，カレント・レコードあるいは相対レコードなどのポインタの管理をユーザが行わなければならず，その手続きは多少面倒なものになります．

また，FCBのところでも述べたように，ファイル名にパス名を使用できないので，階層ディレクトリ構造には対応できないという致命的な欠点ももっています．

MS-DOS ver.2.11では，これらの欠点を補うものとして，UNIX流のファイル・ハンドルによるファイル・アクセスを行うためのシステム・コールも用意されています．

ファイル・ハンドルとは，単に0から始まる整数のことで，ファイル・オープンのシステム・コールが発行されると，このファイル・ハンドル(整数値)が返されます．プログラムでは，以降このファイル・ハンドルを覚えておいて(メモリやレジスタに格納しておく)，この値を指定されたレジスタにセットし，読み書きあるいはクローズなどのシステム・コールを行います．これによって，目的のファイルに対してアクセスが可能となり，非常に簡単にファイルを取り扱うことができます．

なお，プロセスが起動された時点で，すでに**表3-2**(86ページ)に示した五つのファイル・ハンドルはシステムによってオープンされているので，これらの標準入出力に対しては改めてオープンする必要はありません．

このファイル・ハンドルによるファイル・アクセスでは，ファイルは1バイト単位で扱っていて，次にアクセスされるべきファイル内の位置を指すものとして，ファイル・ポインタ(4バイト)と呼ばれる32ビットの符号つき整数が使われています．

ランダム・アクセスする場合は，このファイル・ポインタ移動のシステム・コールを用いてファイル・ポインタを移動し，ファイルの任意の場所をアクセスすることができます．ファイルをクローズすると，そのファイル・ハンドルは解放されて次のファイル・オープンの際に使用されます．

また，ファイル・ハンドルによるシステム・コールではデバイス名も扱えることから，デバイスにおいてもファイルとまったく同様の感覚でアクセスを行うことができます．

ファイル・ハンドルを扱ううえでもう一つ重要な問題は，階層プロセスの場合に，このファイル・ハンドルは子プロセスに継承されるということです．親プロセスでオープンしたファイル・ハンドルは，環境と同様に子プロセスに引き継がれます．よって，子プロセスでは改めてファイルをオープンしなくても，それらのファイルをアクセスできることになります．

また環境と同様に，子プロセスでこれらの継承されたファイルをクローズしても，親プロセスではオープンされた状態が保たれていることにも注意が必要です．

＊　　　　　　＊

この章では，MS-DOSのファイル・アクセスを行うための知識について解説しました．

MS-DOSでは，CP/Mから受け継いだFCBによるファイル・アクセスと，UNIXから受け継いだファイル・ハンドルによるファイル・アクセスの両方のアクセス・モードをサポートしています．このため，MS-DOSのファイル・システムを理解するためには2倍の労力を必要としますが，反面では知識も2倍になるわけですから勉強のやりがいがあります．

FCBによるファイル・アクセスは，CP/M互換の必要性がない限り，現在では使われていない「古い方法」です．しかし，CP/M時代からのソフトウェア資産をもっているユーザは熟知しておく必要があるでしょう．

一方，ファイル・ハンドルによるファイル・アクセスは，UNIX流の「新しい方法」であり，今後のプログラム開発には後者の方法を用いるべきです．

本章で解説したMS-DOSのファイル・システムを熟知することによって，一部の市販品にみられるような，ファイル・クラッシュ(破壊)した際のリカバリ・プログラムを作成するのも難しいことではありません．

第II部

MS-DOSのプログラム・インターフェース

第I部では，主としてMS-DOS上におけるプログラム作成のルールやその方法について解説しました．第II部では，ユーザ・アプリケーションのためにMS-DOSがサービスしているシステム・コールについて解説します．

システム・コール自体の利用方法はマニュアルにも詳しく書かれているため，ここではそのシステム・コールを利用する際の注意点と，実際にアクセスするためのプログラム実例を示して解説しています．

とくにプログラム実例では，C言語とアセンブリ言語を併用し，日本語を用いた詳しいコメントを多用するなど，「プログラムの読みやすさ」に重点をおいて記述しました．

また，これらのプログラムはMS-DOSの強化コマンドとして，実用的にも十分に耐え得るものも含まれています．

ソフトウェアの解説というのは，紙数の制限などもあって，すべてを完全に解説するというのは不可能に近いものがあります．また読む場合でも，書かれていることすべてを完全に吸収するのは難しいでしょう．

このとき，読者としては実際にプログラミングしてみて，何か問題が生じたなら解説書に示されたソース・リストを参照することによって，かなりの知識を身につけることができます．つまり，ソフトウェア解説書では，文章とともに頼りになるのがプログラミング実例なのです．

一般にソース・リストの中には，再帰的プログラミングのテクニックが含まれていたり，データ構造に関する知識や，データ検索に関するテクニックなど，プログラミングのコツが多く含まれているものです．ソース・リストを文章のように読めるようになると，ソフトウェア解説書の利用価値が倍増します．

第5章 MS-DOSの内部割り込み

ターミネートとエラー・ハンドルとディスク・アクセス

本章では，MS-DOSのアプリケーション・プログラムで使用可能な内部割り込みについて，その機能と使いかたを解説していきます．これらの知識は，MS-DOSの機能を最大限に引き出したユーザ・インターフェースを実現し，かつ互換性の高いプログラムを作成するために欠かせません．

5-1 内部割り込み

8086 CPUでは，256個の割り込み処理ルーチンをもつことができ，この割り込みベクタ・テーブルのためにセグメント・アドレス0000Hのオフセット0000H〜03FFH（4バイト×256）が割り当てられています．そして，この割り込みテーブルには，最初の2バイトにオフセット，その後にセグメント・アドレスが入り，8086 CPUのもつ1Mバイト空間内のどのアドレスにもジャンプできるようになっています．

MS-DOSで使用できる内部割り込み

さて，この内部割り込みには8086 CPUを開発したインテル社により，表5-1のようにすでに使用目的が予約されている割り込みがあります．そして，MS-DOSではINT 20H〜3FHまでの32個の割り込みを使用しています．また，INT 80H〜FCHは割り込みルーチンのアドレス・テーブルとして使用されています．それらのうち，表5-2にあげた8個の割り込みタイプはユーザに解放されています．

このうち，INT 22H〜INT 24Hまでの割り込みテーブルは，単なるアドレスの格納場所として使用されており，真の割り込みではありません．後述のように，これらのアドレスの変更は，ファンクション25Hのシステム・コール（Set Vector；162ページ）によって，ユーザのプログラム内での割り込み処理ルーチンに変更することが可能です．したがって，ユーザのプログラム内で使用できる割り込みタイプはINT 20H，21H，25H，26H，27Hおよび40H〜FFHということになります．

しかし，40H〜FFHの割り込みタイプは，システムによっては使われている場合もあるので，それぞれのマシンのマニュアルを参照して確認する必要があります．

〔表5-1〕8086 CPU割り込み予約

割り込みタイプ	機　能
0	除算エラー
1	シングル・ステップ割り込み
2	NMI
3	1バイト型内部割り込み
4	オーバフロー，INTO割り込み
5〜1FH	インテル社予約

5-2 内部割り込みの機能と使いかた

MS-DOSでの内部割り込みの機能とその使いかたについて，順に解説していきます．

〔表5-2〕ユーザ開放割り込みタイプ

割り込みタイプ	機　能
20H	プログラムの終了（メモリ開放）
21H	システム・コール（ファンクション・リクエスト）
22H	プログラム終了アドレス格納
23H	ctrl-C処理アドレス給納
24H	致命的エラー処理アドレス給納
25H	アブソリュートなディスク読み出し
26H	アブソリュートなディスク書き込み
27H	プログラム常駐終了

Program Terminate	INT 20H
機能	プログラムの終了
コール	CS←PSPのセグメント・アドレス
リターン	なし

割り込みタイプ20Hによって，現在のプロセスを終了し制御が親プロセスに戻ります．この内部割り込みはver.1.25との互換性を保つために用意されているものです．

この割り込みを利用する場合には，CSレジスタはPSPのセグメントを指していなければなりません．COMモデルのプロセスの場合は，CSレジスタはPSPのセグメントを指しているので問題ありませんが，EXEモデルの場合，通常CSレジスタはPSPを指していません．

したがって，EXEモデルの場合はCSレジスタがPSPのセグメントを指すようにプログラム内で操作(スタックの操作など)してやらなければなりません．また，この割り込みを実行するまえにサイズを変更したファイルはクローズしなければなりません(ファンクション10H，3EH)．

このような理由から，この割り込みタイプによるプログラム終了は，EXEモデル向きではないので，後述するファンクション4CH(160ページ)を使用するほうがベターです．また，ファンクション4CHはCOMモデルでも利用できるので，ファンクション4CHだけを覚えておけばどのメモリ・モデルでも使えることになり，将来の互換性も保証されることになります．

Function Request	INT 21H
機能	ファンクション・リクエスト
コール	AH←ファンクション番号 他のレジスタ←個々のファンクションで指定された内容
リターン	各ファンクションの解説を参照

AHレジスタに目的のファンクション番号をセットします．詳細については，次章の各ファンクション・リクエストのところで述べることにします．

Terminate Address	INT 22H
機能	プロセス終了アドレス
コール	ユーザ・プログラムからは使用できない

INT 22Hのベクタは，現在実行中のプロセスが終了した場合に戻るべきアドレスを格納しておくために使用され，ユーザ・プログラムがINT 22Hを直接実行することはできません．

なお，この場合に格納される終了アドレスはver.2.11より以前との互換性を保つために用意されていて，ver.3.10以降では，この終了アドレスは設定されるものの参照されることはなく，実際にはPSPに格納された終了アドレス(オフセット0AH)が使用されます．

ctrl-C Exit Address	INT 23H
機能	ctrl-C処理アドレス
コール	ユーザ・プログラムからは使用できない

MS-DOSでは，システム・コールの実行中にctrl-C入力が検出されると，その処理ルーチンに制御が移ります．INT 23Hの割り込みベクタは，この際のctrl-C処理ルーチンのアドレスを格納しておくために用いられており，ユーザ・プログラム内からINT 23Hを実行することはできません．

このctrl-C処理ルーチンは，通常command.comが用意しておき，その処理アドレスは，ユーザ・プログラムを子プロセスとして起動した際に，PSPを介してユーザ・プログラムでも自動的に共有されます．

また，必要に応じてユーザ・プログラム内で独自にctrl-C処理ルーチンをもつことも可能です．ユーザが作成するctrl-C処理ルーチンは，そのルーチン内でシステム・コールを発行することは可能ですが，そこで呼び出したシステム・コールの実行時に再びctrl-Cが入力され，再帰的にctrl-C処理ルーチンが呼び出される可能性があることを考慮しなければなりません．

また，ユーザの作成するctrl-C処理ルーチンは，次のいずれかの方法で終了しなければなりません．

(1) すべてのレジスタ内容を保存してIRET命令を実行する．この場合，ctrl-C処理ルーチンの終了したあと，もとのプログラムは続行される．

(2) CF(キャリ・フラグ)をセットしないでFAR RET命令を実行する．この場合，ctrl-C処理ルーチンの終了後，もとのプログラムは続行される．

(3) CF(キャリ・フラグ)をセットしてFAR RET命令を実行する．この場合，もとのプログラムは終了させられる．

【INT 23Hのサンプル・プログラム】

リスト5-1(ctrl.c)および**リスト5-2**(ctrlsub.asm)は，独自のINT 23Hハンドラを用意して，子プロセスを実行中にctrl-Cが入力された場合に，本当にプログラムを中断するかどうかをたずねるプログラムctrl.exeのソース・リストです．

ここで掲載するプログラム例では，C言語とアセンブリ言語を用いて記述しています．一般に，プログラムをアセンブリ言語だけで作成すると，システム・コールの使用時におけるレジスタ内容の対応がとりやすくなる反面，その構造が理解しにくくなります．システム・コールを使うプログラムをC言語だけで記述することも可能ですが，プログラムの構造がわかりやす

くなる代わりに，CPU レジスタとの対応がわかりにくくなります．ここでは，システム・コールの応用例を示すために，CPU レジスタとのやりとりを行う部分はアセンブリ言語で記述し，そのほかのプログラムはC言語で記述していくことにします．

◆ ctrl.c

リスト5-1 はプログラム ctrl.exe のC言語ソース部分です．同リストにおいて関数 main では，最初にプログラム・タイトルを表示したのち，コマンド・ライン・パラメータの解析を行います．パラメータの解析では，

〔リスト5-1〕
プログラム ctrl.c

```
 1: /*********************************************************************
 2: *
 3: *   機　　能：   ctrl-C 処理ルーチンを用意して子プロセスを実行する
 4: *   割り込み：   INT 23H
 5: *   サ　　ブ：   ctrlsub.asm
 6: *   生　　成：   masm /ML ctrlsub;
 7: *                cl -J -AS ctrl.c ctrlsub
 8: *   使用方法：   ctrl コマンド名　引数　・・・
 9: *
10: *********************************************************************/
11: #include      <stdio.h>
12: #include      <process.h>
13:
14: void main (int, char **);
15: void set_int23 (void);
16: /*********************************************************************
17: *
18: *   関数名：    main (argc, argv)
19: *   機　能：    INT 23H ハンドラを用意して子プロセスを起動する
20: *   入　力：    int argc         ・・・・  コマンドライン・パラメータの数
21: *               char *argv[]     ・・・・  パラメータ文字列へのポインタ
22: *   出　力：    なし
23: *
24: *********************************************************************/
25: void main (argc, argv)
26: int argc;
27: char **argv;
28: {
29:     int st;
30:
31:     if (argc == 1) {
32:         printf ("使用法： ctrl コマンド名　引数　・・¥n");
33:         exit (1);
34:     }
35:     printf ("¥n *** INT 23H ハンドラ Ver.1.1 ***¥n¥n");
36:     set_int23 ();
37:     if ((st = spawnvp (P_WAIT, argv[1], argv + 1)) == -1) {
38:         perror ("子プロセスが起動できません ");
39:         exit (2);
40:     }
41:     putchar ('¥n');
42:     while (--argc) {
43:         printf ("%s ", *++argv);
44:     }
45:     printf ("--- 終了コード = %d¥n", st);
46:     exit (0);
47: }
```

〔リスト5-2〕
プログラム
ctrlsub.asm ①

```
 1: ;*********************************************************************
 2: ;
 3: ;   機　　能：   INT 23H ハンドラの登録と ctrl-C 入力処理
 4: ;   ファンクション：   01H（キーボード入力）
 5: ;                09H（文字列の出力）
 6: ;                25H（割り込みベクタの設定）
 7: ;   生　　成：   masm /ML ctrlsub;
 8: ;
 9: ;*********************************************************************
10:             .MODEL  SMALL, C
11:             .CODE
12: ;*********************************************************************
13: ;
14: ;   ルーチン名：   set_int23
15: ;   機　　能：   INT 23H ハンドラ (ctrl-c) の登録
16: ;   func     ：   25H（割り込みベクタの設定）
17: ;   入　　力：   なし
18: ;   出　　力：   なし
19: ;
20: ;*********************************************************************
```

パラメータの個数を調べ，子プロセスが指定されていなければプログラムの使用方法を表示してエラー・ストップします．

次に，関数(サブルーチン)set_int23によってINT 23Hハンドラ(プロシージャint_23)を割り込みベクタ・テーブルに登録します．そしてMS-Cのライブラリ関数であるspawnvpによって子プロセスを起動します．

子プロセスの処理が終了したらコマンド・ラインを表示して，その子プロセスから返されるプロセス終了コードを表示します．これによって，このctrl.exeは，あるプログラムの終了コードを調べる際にも利用できることになります．

◆ ctrlsub.asm

〔リスト5-2〕
プログラム
ctrlsub.asm ②

```
21: set_int23    PROC
22:              push    ds
23:              mov     dx, OFFSET int_23
24:              mov     ax, @code
25:              mov     ds, ax
26:              mov     ah, 25h
27:              mov     al, 23h
28:              int     21h                          ;中断アドレス(INT23H)の設定
29:              pop     ds
30:              ret
31: set_int23    ENDP
32:
33: ;*********************************************************************
34: ;
35: ;       ルーチン名 :   int_23
36: ;       機    能 :   INT 23H ハンドラ(ctrl-c入力の処理)
37: ;       func     :   01H(キーボード入力)
38: ;                    09H(文字列の出力)
39: ;       入    力 :   なし
40: ;       出    力 :   キャリフラグ
41: ;                    CF=セット :   プログラムの終了
42: ;                    CF=クリア :   プログラムの続行
43: ;
44: ;*********************************************************************
45: int_23       PROC    FAR
46:              push    ax
47:              push    dx
48:              push    ds
49:              mov     ax, @data
50:              mov     ds, ax
51: int23_loop:
52:              mov     dx, OFFSET abort_msg
53:              mov     ah, 09h
54:              int     21h                          ;中断メッセージ
55:              mov     ah, 01h
56:              int     21h                          ;キーボード入力
57:              cmp     al, 'a'
58:              jb      upper
59:              cmp     al, 'z'
60:              ja      upper
61:              sub     al, 'a' - 'A'                ;大文字→小文字変換
62: upper:
63:              cmp     al, 'Y'
64:              je      abort
65:              cmp     al, 'N'
66:              jne     int23_loop
67:              stc
68: abort:
69:              cmc
70:              pop     ds
71:              pop     dx
72:              pop     ax
73:              ret
74: int_23       ENDP
75:
76:              .DATA
77: ;*********************************************************************
78: ;
79: ;       データ名 :   abort_msg
80: ;       機    能 :   プログラム中断メッセージ
81: ;
82: ;*********************************************************************
83: abort_msg    DB      0dh, 0ah
84:              DB      'プログラムを中断しますか <Y/N>?'
85:              DB      '$'
86:              END
```

リスト5-2はプログラムctrl.exeのアセンブリ・ソース部分です．サブルーチン(関数)set_int23は，ファンクション25Hを用いて，ユーザの用意したINT 23Hハンドラ(プロシージャint_23)のエントリ・アドレスを割り込みベクタ・テーブルに登録します．

子プロセスの実行中にctrl-Cが入力されるとプロシージャint_23に制御が移ります．プロシージャint_23では，まず中断メッセージを表示してキーボード入力待ちとなります．もし，キーボードから“y”が入力されるとCF(キャリ・フラグ)をセットしてFAR RETします．これによって子プロセスは終了させられることになります．また，“n”が入力されたらCFをクリアしてFAR RETします．これによって子プロセスはその処理が続行されます．

◆ 生成方法

プログラムctrl.exeは，次の手順で分割アセンブル/コンパイルして作成します．

masm /ML ctrlsub;

cl －J －As ctrl.c ctrlsub

マクロ・アセンブラMASMの/MLオプションは，Cソースに合わせて大文字と小文字の区別を行います．clコマンドでは，ctrl.cをコンパイルし，あらかじめアセンブルしてあるctrlsub.objとリンクを行います．

ここで，clコマンドはデフォルトで－Asオプション，すなわちメモリ・モデルとしてスモール・モデルを選択しますが，ここで紹介したプログラムはスモール・モデル用なので明示的に－Asオプションを指定しています．

〔リスト5-3〕
プログラム ctrl.exe の実行例

```
R>dump ctrl.exe ⏎… dumpコマンドでファイル ctrl.exe をダンプする①

Dump Version 2.1

00000000  4D 5A C0 00 13 00 07 00-20 00 C3 00 FF FF 6F 02   MZﾀ.........ﾃ...o.
00000010  00 08 8A 1D 39 02 00 00-1E 00 00 00 01 00 BF 00   ....9.........ｿ.
00000020  00 00 D1 00 00 00 A2 01-C5 01 50 02 00 00 85 06   ..ﾑ...｢.ﾅ.P.....
00000030  00 00 2E 04 C5 01 32 04-C5 01 00 00 00 00 00 00   ....ﾅ.2.ﾅ.......
00000040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000060  00 00 00 00 00 00 0^C … ctrl-C を入力する②
R>ctrl dump ctrl.exe ⏎INT 23Hハンドラを用意して dump コマンドを実行する③

 *** INT 23H ハンドラ Ver.1.1 ***

Dump Version 2.1

00000000  4D 5A C0 00 13 00 07 00-20 00 C3 00 FF FF 6F 02   MZﾀ.........ﾃ...o.
00000010  00 08 8A 1D 39 02 00 00-1E 00 00 00 01 00 BF 00   ....9.........ｿ.
00000020  00 00 D1 00 00 00 A2 01-C5 01 50 02 00 00 85 06   ..ﾑ...｢.ﾅ.P.....
00000030  00 00 2E 04 C5 01 32 04-C5 01 00 00 00 00 00 00   ....ﾅ.2.ﾅ.......
00000040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000080  00 00 00 00 00 00 00 ^C … ctrl-C を入力する④
   ┌─中断するかどうか聞いてくる⑤      ┌─n を入力して dump コマンドを続行⑥
プログラムを中断しますか <Y/N>?n00-00 00 00 00 00 00 00 00   ................
00000090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000180  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000190  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000001A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000001B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 ^C
   ┌─中断メッセージ                                     └─再び ctrl-C を入力⑦
プログラムを中断しますか <Y/N>?y ←─今度は y を入力して中断する⑧
dump ctrl.exe --- 終了コード = 256←─プロセス終了コード
R>
  └─コマンド・ラインの表示⑨
```

－Jオプションは，2バイト・コード(シフトJISコード)を日本語として正しく認識させるためのコンパイル・オプションです．

◆ **実行サンプル**

リスト5-3はプログラムctrl.exeの実行例を示しています．同リストではctrl.exeの子プロセスとしてdumpコマンドを起動し，ファイル・ダンプの途中でctrl-Cを入力してテストしています．

① まず，ctrl.exeを使用しないでdumpコマンドを起動する．

② 途中でctrl-Cを入力すると直ちにdumpコマンドを終了する．

③ 次に，ctrl.exeの子プロセスとしてdumpコマンドを指定して起動する．このときdumpコマンドに渡すファイル名も入力できる．

④ ファイル・ダンプの途中でctrl-Cを入力する．

⑤ すると，プロシージャint_23に制御が移り，プログラムを中断するかどうかを聞いてくる．

⑥ "n"を入力してdumpコマンドを続行する．

⑦ 再びctrl-Cを入力する

⑧ やはり，プログラムを中断するかどうかを聞いてくるので，今度は"y"を入力する．

⑨ すると，子プロセス(dump.exe)を終了し，ctrl.c内の関数mainに処理が戻り，コマンド・ライン表示が行われる．

⑩ 同時に，子プロセス(dump.exe)から返されたプロセス終了コードが表示される．

Critical Error Handler Address	INT 24H
機能	致命的エラー処理アドレス
コール	ユーザ・プログラムからは使用できない

MS-DOSでは，システム・コール実行中に致命的なエラー(たとえばディスク・アクセス・エラーなど)が発生した際に，そのエラー処理ルーチンに制御が移ります．INT 24Hの割り込みベクタは，この際のエラー処理ルーチンのアドレスを格納しておくために用いられ，ユーザ・プログラム内からINT 24Hを直接実行することはできません．

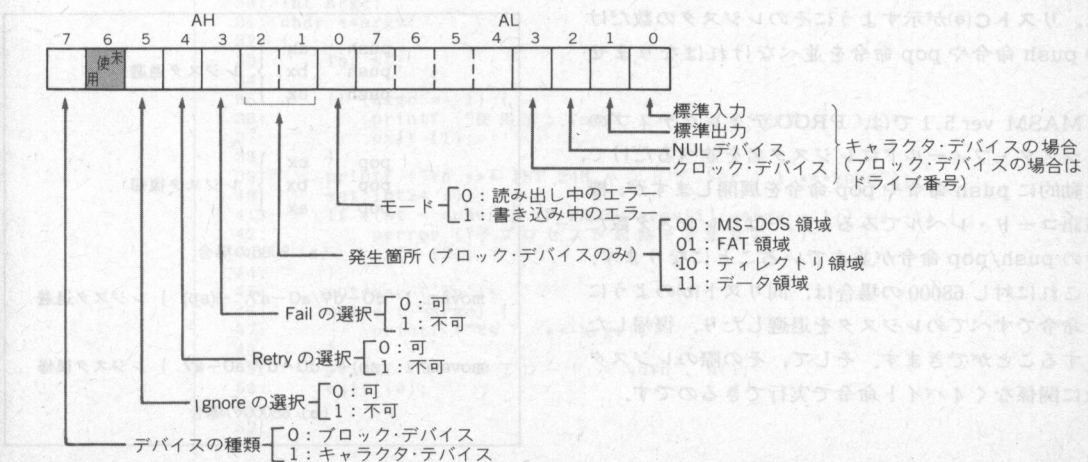
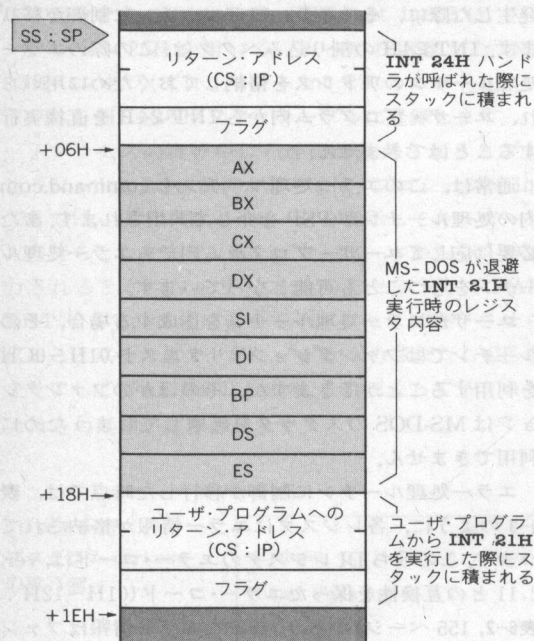
通常は，このエラー処理ルーチンもcommand.com内の処理ルーチンがPSPを介して共用されます．また必要に応じてユーザ・プログラム内でもエラー処理ルーチンをもつことも可能となっています．

ユーザがエラー処理ルーチンを作成する場合，そのルーチンではファンクション・リクエスト01H～0CHを利用することができますが，そのほかのファンクションはMS-DOSのスタックを破壊してしまうために利用できません．

エラー処理ルーチンに制御が移行した時点では，**表5-3**のように，各レジスタにエラー情報が格納されています．このうちDIレジスタのエラー・コードは，ver. 2.11との互換性を保ったエラー・コード(01H～12H；**表6-2**，155ページ)であり，詳細なエラー情報はファンクション59H(Get Extended Error；164ページ)によって得ることができます．AXレジスタに返される制御に関する情報は**図5-1**のような意味をもち，これらのビットによりエラーに対する動作の選択が制限されます．

〔表5-3〕INT 24Hでの各レジスタの内容

レジスタ	内容
AH	INT 24Hハンドラの制御に関する情報
AL	エラーが発生したドライブ番号
DI	エラー・コード
BP SI	エラーの発生したデバイスのデバイス・ヘッダへのポインタ

〔図5-1〕AXレジスタの内容

〔図5-2〕INT 24H ハンドラ起動時のスタック内容

また，エラー処理ルーチンが起動されたときのスタックは，**図5-2**のスタック・フレームで構成されます．ここで，エラー処理ルーチンから MS-DOS に戻らずにユーザ・プログラムに戻るには，スタック・トップの IP，CS，FLAG を捨て，各レジスタ内容を POP してから IRET 命令を実行します．これによって，エラーの発生した INT 21H 命令(ユーザ・プログラムからのファンクション・リクエスト)の次の命令に戻ることができます．

エラー処理ルーチンでは，SS，SP，DS，ES，BX，CX，DX の各レジスタ内容は保存しなければなりません．そして，エラー処理ルーチンから戻るには，AL レジスタに下記の値を設定してから IRET 命令を実行します．MS-DOS は，このエラー処理ルーチンから返される AL レジスタの値を参照して処理の選択を行います．

(1) AL＝00H の場合(Ignore)

この場合エラーは無視され，もとのプログラムはそのまま続行されます．Ignore を選択できないのに Ignore を選択すると Fail とみなされます．

(2) AL＝01H の場合(Retry)

この場合は再試行されます．再試行にも失敗すると再度エラーが発生します．Retry を選択できないのに Retry を選択すると Fail とみなされます．

(3) AL＝02H の場合(Abort)

この場合，ユーザ・プログラムには戻らずにユーザ・プログラム(プロセス)を終了します．Abort は制限なく選択が可能です．

(4) AL＝03H の場合(Fail)

この場合，システム・コールが失敗したものとしてユーザ・プログラムにエラーが返されます．ユーザ・プログラム側では，ファンクション 59H によってエラーの詳細な情報を得ることができます．Fail を選択できないのに Fail を選択すると Abort とみなされます．

● 8086 vs 68000(その3) レジスタの保存 ●

8086 CPU では，レジスタの退避や復帰を行う際に，**リストC**(a)が示すようにそのレジスタの数だけの push 命令や pop 命令を並べなければなりません．

MASM ver.5.1 では，PROC ディレクティブのレジスタ・フィールドにレジスタ名を並べるだけで，自動的に push 命令や pop 命令を展開しますが，機械語コード・レベルでみると，やはりレジスタ数だけの push/pop 命令が並んでいることになります．

これに対し 68000 の場合は，同リスト(b)のように1命令ですべてのレジスタを退避したり，復帰したりすることができます．そして，その際のレジスタ数に関係なく4バイト命令で実行できるのです．

〔リストC〕レジスタ保存の比較

```
push  ax  }
push  bx  } レジスタ退避
push  cx  }
  :
pop   cx  }
pop   bx  } レジスタ復帰
pop   ax  }
```

(a) 8086の場合

```
movem.l  d0-d7/a0-a7, -(sp)  } レジスタ退避
  :
movem.l  (sp)+, d0-d7/a0-a7  } レジスタ復帰
```

(b) 68000の場合

【INT 24H のサンプル・プログラム】

リスト5-4(fatal.c)および**リスト5-5**(fatalsub.asm)は，独自のINT 24Hハンドラを用意し，あるプログラムを実行中にディスク・エラーなどの致命的エラーが発生した場合に，その詳しいエラー情報を表示して処理の選択を促すプログラムfatal.exeのソース・リストです．

◆ fatal.c

リスト5-4はプログラムfatal.exeのCソース部分です．同リストにおいて関数mainでは，最初にプログラム・タイトルを表示したのち，コマンド・ライン・パラメータの解析を行います．パラメータの解析では，コマンド・ライン・パラメータの個数を調べ，もし子プロセスが指定されていなければ，プログラムfatal.exeの使用方法を表示してエラー・ストップします．

次に関数(サブルーチン)set_int24によってINT 24Hハンドラ(int_24)を割り込みベクタ・テーブルに登録します．そして，MS-Cのライブラリ関数spawnvpを用いて，コマンド・ラインで指定された子プロセスを起動します．子プロセスの処理が終了したら，起動時に指定されたコマンド・ライン・パラメータを表示して，その子プロセスから返されたプロセス終了コードを表示します．

◆ fatalsub.asm

リスト5-5はプログラムfatal.exeのアセンブリ・ソース部分です．サブルーチン(関数)set_int24は，ファンクション25Hを用いてここで用意したINT 24Hハンドラ(プロシージャint_24)のエントリ・アドレスを割り込みベクタ・テーブルに登録します．

子プロセスの実行中に，ディスク・エラーなどの致

〔リスト5-4〕
プログラム
fatal.c ①

```
 1: /*******************************************************************
 2: *
 3: *    機    能 :  致命的エラー処理ルーチンを用意して子プロセスを実行する
 4: *    割り込み :  INT 24H
 5: *    サ    ブ :  fatalsub.asm
 6: *    生    成 :  masm /ML fatalsub;
 7: *                cl -J -AS -Gs fatal.c fatalsub
 8: *    使用方法 :  fatal コマンド名  引数  ・・
 9: *
10: *******************************************************************/
11: #include    <stdio.h>
12: #include    <process.h>
13: #include    <memory.h>
14: #define     STR_LEN      100
15: int      c;
16:
17: void main (int, char **);
18: void set_int24 (void);
19: void int24_msg (unsigned int, unsigned int, char *);
20: /*******************************************************************
21: *
22: *    関数名 :  main (argc, argv)
23: *    機  能 :  INT 24H ハンドラを用意して子プロセスを起動する
24: *    入  力 :  int argc      ·····  コマンドライン・パラメータの数
25: *              char *argv[]  ·····  パラメータ文字列へのポインタ
26: *    出  力 :  なし
27: *
28: *******************************************************************/
29: void main (argc, argv)
30: int argc;
31: char **argv;
32: {
33:     int st;
34:
35:     if (argc == 1) {
36:         printf ("使用法 : fatal コマンド名  引数  ··¥n");
37:         exit (1);
38:     }
39:     printf ("¥n *** INT 24H ハンドラ Ver.1.1 ***¥n¥n");
40:     set_int24 ();
41:     if ((st = spawnvp (P_WAIT, argv[1], argv + 1)) == -1) {
42:         perror ("子プロセスが起動できません  ");
43:         exit (2);
44:     }
45:     putchar ('¥n');
46:     while (--argc) {
47:         printf ("%s ", *++argv);
48:     }
49:     printf ("--- 終了コード  = %d¥n", st);
50:     exit (0);
51: }
52:
```

〔リスト5-4〕
プログラム
fatal.c ②

```
 53: /*********************************************************************
 54: *
 55: *    関数名：  int24_msg (err, code, ptr)
 56: *    機　能：  エラーメッセージの表示
 57: *    入　力：  err  ‥‥ ＡＸレジスタに返されるエラー情報
 58: *              code ‥‥ ＤＩレジスタに返されるエラー・コード
 59: *              ptr  ‥‥ デバイス名へのポインタ
 60: *    出　力：  なし
 61: *
 62: *********************************************************************/
 63: void int24_msg (err, code, ptr)
 64: unsigned int err, code;
 65: char *ptr;
 66: {
 67:     int aa;
 68:
 69:     if (err & 0x8000) {
 70:         disp ("¥nデバイス  ");
 71:         disp (ptr);
 72:         disp (" ");
 73:
 74:     } else {
 75:         c = (err & 0x000F) + 'A';
 76:         disp ("¥n¥x07ドライブ  ");
 77:         disp (&c);
 78:         disp (" の ");
 79:
 80:         switch ((err & 0x0600) >> 9) {
 81:         case 0:
 82:             disp ("MS-DOS"); break;
 83:         case 1:
 84:             disp ("FAT"); break;
 85:         case 2:
 86:             disp ("ディレクトリ"); break;
 87:         case 3:
 88:             disp ("データ"); break;
 89:         }
 90:         disp ("領域");
 91:     }
 92:     switch ((err & 0x0100) >> 8) {
 93:     case 0:
 94:         disp ("の読み出し"); break;
 95:     case 1:
 96:         disp ("への書き込み"); break;
 97:     }
 98:     disp ("中に致命的エラーが発生しました．¥n");
 99:     disp ("その内容は  ----");
100:     switch (code) {
101:     case 0:
102:         disp ("書き込み禁止です．¥n"); break;
103:     case 1:
104:         disp ("存在しないユニットです ¥n"); break;
105:     case 2:
106:         disp ("ドライブの準備ができていません．¥n"); break;
107:     case 3:
108:         disp ("存在しないファンクション・コードです．¥n"); break;
109:     case 4:
110:         disp ("データのCRCエラーです．¥n"); break;
111:     case 5:
112:         disp ("コマンド・パケットの長さがちがいます．¥n"); break;
113:     case 6:
114:         disp ("シーク・エラーです．¥n"); break;
115:     case 7:
116:         disp ("メディアのタイプがちがいます．¥n"); break;
117:     case 8:
118:         disp ("セクタが存在しません．¥n"); break;
119:     case 9:
120:         disp ("プリンタの用紙切れです．¥n"); break;
121:     case 10:
122:         disp ("書き込みできません．¥n"); break;
123:     case 11:
124:         disp ("読み込みできません．¥n"); break;
125:     case 12:
126:         disp ("ディスク不良です．¥n"); break;
127:     }
128:     disp ("つぎの処理を選択して下さい¥n¥n");
129:     disp ("A  ‥‥  処理中断¥n");
130:     disp ("R  ‥‥  再試行¥n");
131:     disp ("I  ‥‥  処理続行¥n");
132:     disp ("F  ‥‥  処理失敗¥n");
133:     disp ("どれを選択しますか：");
134: }
```

命的エラーが発生すると，プロシージャ int_24 に制御が移ります．プロシージャ int_24 では，サブルーチン name_copy を用いてエラーの発生したデバイス名の読み出しを行い(キャラクタ・デバイスのみ有効)，そのデバイス名へのポインタ，および DX, AX レジスタに返されたエラー・コードを引数として関数 fatal_msg を呼び出し，詳しいエラー情報の表示を行います．

次に，サブルーチン keyin によってキー入力を促し，ユーザの選択に合わせて AL レジスタを設定して IRET 命令を実行し，エラーの発生したアドレス(システム・コール内のルーチン)に戻ります．

ここでプロシージャ int_24 が呼び出された時点では，SS：SP レジスタは，MS-DOS のスタック領域を指していて，MS-C(関数 main)によって確保されたスタック領域とはまったく異なる領域を指しているので，C ソース部分をコンパイルする際に“−Gs”オプションを指定して，スタック・チェック・コードの生成を抑止しないと，“stack overflow”のエラー・メッセージを表示して fatal.exe を中断してしまいます．

また ES, DS の各セグメント・レジスタは，関数 main が設定したデータ・エリア(@data)を指していなければ，データ・バッファ(info)や文字列定数などが正しくアクセスできないので注意が必要です．このような理由から，関数 fatal_msg 内では，printf や scanf などの MS-C に標準のライブラリ関数を使用することができない(すでにスタック・チェック・コードが含まれている)ので，文字列の表示にはサブルーチン(関数)disp を用意して処理しています．

◆ 生成方法

プログラム fatal.exe は，次の手順で分割アセンブル/コンパイルして作成します．

```
masm /ML fatalsub;
cl −J −As −Gs fatal.c fatalsub
```

◆ 実行サンプル

リスト5-6 はプログラム fatal.exe の実行例を示しています．同リストでは，プログラム fatal.exe の子プロセスとして chkdsk コマンドを起動し，ドライブのドアを開けて致命的エラーを故意に発生させています．

① まず，ドライブのドアを開けておいて chkdsk コマンドを起動する．

② すると，command.com 内の INT 24H ハンドラに制御が移り，処理の選択を促す．

③ ここでは“a”を入力して処理を中断する．

④ 次に，fatal.exe を用いてその子プロセスとして chkdsk コマンドを起動する．ここではドライブのドアを閉じて正常に動作することを確認する．

⑤ 返された終了コードから正常に終了したことが確認される．

⑥ 次に，もう一度 fatal.exe を用いて chkdsk コマン

〔リスト5-5〕プログラム fatalsub.asm ①

```
 1: ;*****************************************************************
 2: ;
 3: ;     機    能 :   INT 24H ハンドラの登録と致命的エラー処理
 4: ;     ファンクション :   01H (文字の入力)
 5: ;                  02H (文字の表示)
 6: ;                  25H (割り込みベクタの設定)
 7: ;     生    成 :   masm /ML fatalsub;
 8: ;
 9: ;*****************************************************************
10:             .MODEL  SMALL, C
11:             .CODE
12:             EXTRN   int24_msg:near
13: ;*****************************************************************
14: ;
15: ;     ルーチン名 :  set_int24
16: ;     機    能 :  INT 24H ハンドラ (エラー処理ルーチン) の登録
17: ;     func      :  25H (割り込みベクタの設定)
18: ;     入    力 :  なし
19: ;     出    力 :  なし
20: ;
21: ;*****************************************************************
22: set_int24   PROC
23:             push    ds
24:             mov     dx, @code           ;コード・セグメントとの整合
25:             mov     ds, dx
26:             mov     dx, OFFSET int_24
27:             mov     ah, 25h
28:             mov     al, 24h
29:             int     21h                 ;エラー処理アドレス (INT24H) の設定
30:             pop     ds
31:             ret
32: set_int24   ENDP
33:
```

〔リスト5-5〕プログラム fatalsub.asm ②

```
 34: ;****************************************************************
 35: ;
 36: ;        ルーチン名 :   keyin
 37: ;        機   能 :   キーボード入力と大文字への変換
 38: ;        func    :   01H（キーボード入力）
 39: ;        入   力 :   なし
 40: ;        出   力 :   ax ← 文字コード（英字は大文字に変換）
 41: ;
 42: ;****************************************************************
 43: keyin       PROC
 44:             mov     ah, 01h
 45:             int     21h                     ;キーボード入力
 46:             cmp     al, 'a'
 47:             jb      upper
 48:             cmp     al, 'z'
 49:             ja      upper
 50:             sub     al, 'a' - 'A'           ;小文字→大文字変換
 51: upper:
 52:             xor     ah, ah
 53:             ret
 54: keyin       ENDP
 55:
 56: ;****************************************************************
 57: ;
 58: ;        ルーチン名 :   disp
 59: ;        機   能 :   文字列の表示
 60: ;        func    :   02H（文字の出力）
 61: ;        入   力 :   arg1  文字列へのポインタ
 62: ;        出   力 :   なし
 63: ;
 64: ;****************************************************************
 65: disp        PROC    arg1:PTR
 66:             push    ax
 67:             push    bx
 68:             push    dx
 69:             mov     bx, arg1                ;ポインタ
 70:             mov     ah, 02h                 ;ファンクション02H
 71: disp_loop:
 72:             mov     dl, [bx]
 73:             or      dl, dl                  ;'¥0'（文字列の終端）?
 74:             je      disp_exit
 75:             cmp     dl, 0Ah                 ;LFコード?
 76:             jne     disp_next
 77:             mov     dl, 0Dh                 ;CRコード
 78:             int     21h
 79:             mov     dl, 0Ah
 80: disp_next:
 81:             int     21h
 82:             inc     bx
 83:             jmp     disp_loop
 84: disp_exit:
 85:             pop     dx
 86:             pop     bx
 87:             pop     ax
 88:             ret
 89: disp        ENDP
 90:
 91: ;****************************************************************
 92: ;
 93: ;        ルーチン名 :   name_cpy
 94: ;        機   能 :   デバイス名の読み出し
 95: ;        入   力 :   なし
 96: ;        出   力 :   なし
 97: ;
 98: ;****************************************************************
 99: name_copy   PROC
100:             push    ax
101:             push    cx
102:             push    si
103:             push    di
104:             push    ds
105:             push    es
106:
107:             push    ds
108:             pop     es
109:             mov     di, OFFSET name_buf
110:             add     si, 10
111:             mov     ds, bp
112:             mov     cx, 8
113:             cld
```

〔リスト5-5〕プログラム fatalsub.asm ③

```
114:            rep     movsb
115:            mov     BYTE PTR es:[di+1], 0       ;文字列の終端('¥0')
116:
117:            pop     es
118:            pop     ds
119:            pop     di
120:            pop     si
121:            pop     cx
122:            pop     ax
123:            ret
124: name_copy  ENDP
125:
126: ;*********************************************************************
127: ;
128: ;     ルーチン名:  int_24
129: ;     機   能:  INT 24H ハンドラ(致命的エラーの処理)
130: ;     入   力:  なし
131: ;     出   力:  ax=00h → Ignore(処理続行)
132: ;              ax=01h → Retry(再試行)
133: ;              ax=02h → Abort(処理中断)
134: ;              ax=03h → Fail(処理失敗)
135: ;
136: ;*********************************************************************
137: int_24     PROC    FAR
138:            push    bx
139:            push    cx
140:            push    dx
141:            push    ds
142:            push    es
143:            mov     cx, @data;
144:            mov     ds, cx
145:            mov     es, cx                      ;ES,DSレジスタを設定
146:            mov     info, ah                    ;AHレジスタのエラー情報格納
147:            call    name_copy                   ;デバイス名の読み出し
148: int24_loop:
149:            mov     cx, OFFSET name_buf
150:            push    cx                          ;デバイス名へのポインタ(ptr)
151:            push    di                          ;エラー・コード(code)
152:            push    ax                          ;エラー情報(err)
153:            call    int24_msg                   ;エラー・メッセージの表示
154:            add     sp, 6                       ;スタックの整合
155:            call    keyin                       ;キー入力
156:            cmp     al, 'A'
157:            mov     ah, 2
158:            je      int24_exit                  ;中断
159:            mov     ah, info
160:            and     ah, 10h
161:            cmp     ax, 'R' OR (10h SHL 8)
162:            mov     ah, 1
163:            je      int24_exit                  ;再試行
164:            mov     ah, info
165:            and     ah, 20h
166:            cmp     ax, 'I'
167:            mov     ah, 0
168:            je      int24_exit                  ;処理続行
169:            mov     ah, info
170:            and     ah, 8h
171:            cmp     ax, 'F'
172:            mov     ah, 3
173:            jne     int24_loop                  ;処理失敗
174: int24_exit:
175:            mov     al, ah
176:            pop     es
177:            pop     ds
178:            pop     dx
179:            pop     cx
180:            pop     bx
181:            iret
182: int_24     ENDP
183:
184:            .DATA
185: ;*********************************************************************
186: ;
187: ;     データ名:  info
188: ;     機   能:  エラー情報のバッファ
189: ;
190: ;*********************************************************************
191: info       DB      ?
192: name_buf   DB      10      dup (?)
193:            END
```

ドを起動する．ここではドライブのドアを開けて致命的エラーを発生させる．

⑦ ユーザの用意したINT 24Hハンドラ(int_24)に制御が移り，エラー処理メッセージが表示され処理の選択を促す．

⑧ "r" を選択し再試行を試みる．

⑨ 同様にユーザの用意したエラー処理メッセージが表示される．ここでは "i" を選択してみる．

⑩ chkdskコマンドからのエラー・メッセージが表示され，chkdskコマンドは終了する．

⑪ chkdskコマンドから返されたプロセス終了コードであり，異常終了したことがわかる．

⑫ 再びfatal.exeを用いてchkdskコマンドを起動する．ここでもドライブのドアは開けたままとする．

⑬ ユーザの用意したエラー処理メッセージが表示されるので，"a" を選択して処理の中断を行う．

〔リスト5-6〕プログラム fatal.exe の実行例

```
R>chkdsk ⏎… ドライブのドアを開けて chkdsk コマンドを実行①

ドライブの準備ができていません．<読み取り中><ドライブ B:>  ┐
中止<A>，もう一度<R>，無視<i>? a ⏎… 中止する③          ┘ command.com のエラー処理②

>fatal chkdsk b: ⏎ … INT 24H ハンドラを用意して chkdsk コマンドを実行（ドアは閉めておく）④

 *** INT 24H ハンドラ  Ver.1.1 ***

   1250304 バイト  : 全ディスク容量
     61440 バイト  : 2 個のシステムファイル
      2048 バイト  : 1 個のディレクトリ
    677888 バイト  : 49 個のユーザーファイル
    508928 バイト  : 使用可能ディスク容量

    655360 バイト  : 全メモリ
    277664 バイト  : 使用可能メモリ

chkdsk b: --- 終了コード  = 0 ←---正常終了⑤

R>fatal chkdsk b: ⏎… もう一度 INT 24H ハンドラを用意して chkdsk コマンドを実行する（ドアは開ける）⑥

 *** INT 24H ハンドラ  Ver.1.1 ***

ドライブ B の FAT領域の読み出し中に致命的エラーが発生しました.  ┐
その内容は  ----  ドライブの準備ができていません.             │
つぎの処理を選択して下さい                                    │
                                                              │
A  ····  処理中断                                             ├ ユーザ (fatal.exe) の用意したエラー処理⑦
R  ····  再試行                                               │
I  ····  処理続行                                             │
F  ····  処理失敗                                             │
どれを選択しますか : r ⏎… 再試行⑧                            ┘
ドライブ B の FAT領域の読み出し中に致命的エラーが発生しました.
その内容は  ----  ドライブの準備ができていません.
つぎの処理を選択して下さい

A  ····  処理中断
R  ····  再試行
I  ····  処理続行           ┌─無視する⑨
F  ····  処理失敗           ↓
どれを選択しますか : i      ドライブの指定が違います. ←──chkdsk コマンドから出力されたエラー・メッセージ⑩

chkdsk b: --- 終了コード  = 255 ←──異常終了⑪

R>fatal chkdsk b: ⏎ … もう一度 INT 24H ハンドラを用意して chkdsk コマンドを実行（ドアは開けたまま）⑫

 *** INT 24H ハンドラ  Ver.1.1 ***

ドライブ B の FAT領域の読み出し中に致命的エラーが発生しました.
その内容は  ----  ドライブの準備ができていません.
つぎの処理を選択して下さい

A  ····  処理中断
R  ····  再試行
I  ····  処理続行
F  ····  処理失敗
どれを選択しますか : a ⏎… 中断の選択⑬
chkdsk b: --- 終了コード  = 512 ←──異常終了⑭

R>
```

⑭ chkdskコマンドから返された終了コードであり，やはり異常終了したことを表している．

Absolute Disk Read	INT 25H
機能	ディスク・セクタの直接読み出し
コール	AL ←ドライブ番号 (00H＝A，01H＝B，…) DS：BX ←ディスク転送アドレス CX ←読み込みセクタ数 DX ←読み込み開始セクタ
リターン	CF＝1　エラー発生 AL レジスタにエラー・モードを返す CF＝0 正常終了

この内部割り込みでは，ディスク上の指定された論理セクタから指定されたセクタ数だけのデータを読み込み，転送アドレスで指定したメモリ領域にそのデータを格納します．

この内部割り込みでは，セグメント・レジスタ以外のすべてのレジスタは破壊されます．また，この割り込み処理中に致命的エラーが生じても，その処理ルーチンに飛ぶことはせず，AL レジスタにエラー・コードをセットして返します(エラー・コードについては第6章参照)．

この割り込みタイプをコールする際にフラグ(エラー情報)がスタックに積まれますが，ユーザ・プログラムに戻った時点でもこのフラグはスタックに残ったままになっています(結果がフラグに返される)．よって，ユーザ・プログラム内では，このフラグを参照したら，POPF 命令を実行してスタック・ポインタを合わせてやらなければなりません．

【INT 25H のサンプル・プログラム】

リスト5-7(ddump.c)および**リスト5-8**(ddumpsub.asm)は，INT 25H の内部割り込みを利用して，指定されたドライブの指定された論理セクタを読み出して，メモリ・ダンプ・フォーマットで表示するプログラム ddump.exe のソース・リストです．

◆ **ddump.c**

リスト5-7 はプログラム ddump.exe の C ソース部分です．同リストにおいて構造体 _DSK は，ファンクション 36H(Get Disk Free Space；166 ページ)によって得られるセクタ数やクラスタ数などのディスクに関する情報を格納するデータ・ブロックの構造を定義しています．

関数 main では，最初にプログラム・タイトルを表示したのち，コマンド・ライン・パラメータの解析を行います．パラメータの解析では，−s および−n オプションの場合には，ライブラリ関数 atoi を用いて開始セクタやセクタ数などをそれぞれの変数に格納します．それ以外のオプションが指定されている場合は，その旨を表示してエラー・ストップします．パラメータがドライブ名の場合には，そのドライブ名へのポインタをポインタ drv_name に設定します．

次に，ポインタ drv_name＝＝NULL の場合は，ドライブの指定が省略されているので関数(サブルーチン)func_19 を用いてカレント・ドライブのドライブ番号を調べ，変数 drv_num に格納しておきます．もし，ドライブ名の指定が行われている場合(drv_name!＝NULL)は，そのドライブ名をドライブ番号に変換して，やはり変数 drv_num に格納しておきます．

これらの前処理が終わったら，関数 ddump にドライブ番号，開始セクタ，セクタ数などを渡してディスクのダンプを行います．

関数 ddump では，関数(サブルーチン)func_36 を用いて指定されたドライブのセクタ数やクラスタ数を読み出し，そのディスク情報を指定されたデータ・ブロック(disk_data)に格納します．ここで，もしドライブの指定が間違っていると，構造体のメンバ ax に 0FFFFH が返されるので，そのメンバ ax を調べて 0FFFFH が格納されている場合はエラー・ストップします．

次に関数 disk_msg を呼び出し，得られたディスク・データの表示を行います．これらの処理が終わったらライブラリ関数 calloc を用いてセクタ・バッファの確保を行います．

次に，while ループを用いて関数 int 25 によって指定されたセクタの読み出しを行います．ここで，もし読み出しエラーが発生した場合は，変数 st にエラー・コードが返されるので，エラー・メッセージを表示して ddump.exe を終了します．もし，セクタが正常に読み込まれた場合には，関数 dump を用いてダンプ・リストの表示を行います．この while ループは指定されたセクタの数だけ繰り返され，すべてのセクタのダンプが終了したらプログラム ddump.exe を終了します．

関数 disk_msg は，ドライブ番号 drv_num とディスク・データ構造体へのポインタ ptr を引数として受け取り，ディスク・データの表示を行います．

関数 dump は，ポインタ buff で指定されたメモリ領域の内容を byte で指定されたバイト数のダンプ・リストとして出力します．この際にメモリ・ダンプは 256 バイト単位で行われます．

関数 para_dump は，16 バイト単位でのメモリ・ダンプを行います．ここでは最初にオフセット・アドレス，次にメモリ内容の 16 進表示，その後にメモリ内容に対応した ASCII キャラクタを表示します．

◆ **ddumpsub.asm**

リスト5-8はプログラムddump.exeのアセンブリ・ソース部分です．ストラクチャ_DSKは，**リスト5-7**のddump.cで定義された構造体_DSKに対応し，ディスク・データを格納する領域のデータ構造を定義しています．サブルーチン(関数)func_19は，ファンクション19H(Get Current Disk；165ページ)を用いてカレント・ドライブ番号の取得を行い，そのドライブ

〔リスト5-7〕プログラムddump.c ①

```
 1: /*******************************************************************
 2: *
 3: *   機    能 :   ディスク・ダンプ・プログラム
 4: *   割り込み :   INT 25H
 5: *   サ    ブ :   ddumpsub.asm
 6: *   生    成 :   masm /ML ddumpsub;
 7: *                cl -J -AS ddump.c ddumpsub
 8: *   使用方法 :   ddump [<d:>] [-s<開始セクタ>] [-n<セクタ数>]
 9: *
10: ********************************************************************/
11: #include     <stdio.h>
12: #include     <malloc.h>
13:
14: /*******************************************************************
15: *
16: *   構造体 :     DSK
17: *   機  能 :     ディスク・データ構造の定義
18: *
19: ********************************************************************/
20: typedef struct _DSK {
21:     unsigned int ax;          /* 1クラスタあたりのセクタ数 */
22:     unsigned int bx;          /* 使用可能なクラスタ数 */
23:     unsigned int cx;          /* 1セクタあたりのバイト数 */
24:     unsigned int dx;          /* 1ドライブあたりのクラスタ数 */
25: } DSK;
26:
27: void main (int, char **);
28: void ddump (int, int, int);
29: void disk_msg (int, DSK *);
30: void dump (char *, unsigned int);
31: void para_dump (long, char *);
32: int  func_19 (void);                  /* カレント・ドライブ番号の取得 */
33: void func_36 (int, DSK *);   /* ディスク残り容量の読み出し */
34: int  int_25 (int, char *, int, int);   /* セクタの直接読み出し */
35: /*******************************************************************
36: *
37: *   関数名 :   main (argc, argv)
38: *   機  能 :   コマンドライン・パラメータの解析
39: *   入  力 :   int argc       ･･･  コマンドライン・パラメータの数
40: *              char *argv[]   ･･･  パラメータ文字列へのポインタ
41: *   出  力 :   なし
42: *
43: ********************************************************************/
44: void main (argc, argv)
45: int argc;
46: char *argv [];
47: {
48:     char *drv_name;
49:     int sec_begin, sec_num, drv_num;
50:
51:     printf ("¥n *** ディスク・ダンプ・プログラム Ver.1.1 ***¥n¥n");
52:     drv_name = NULL;
53:     sec_begin = 0;
54:     sec_num = 1;
55:     while (--argc > 0) {
56:         if (**++argv == '-') {
57:             if (toupper (argv[0][1])) {
58:                 tolower (argv[0][1]);
59:             }
60:             switch (argv[0][1]) {
61:             case 's' :
62:                 sec_begin = atoi (*argv + 2); break;
63:
64:             case 'n' :
65:                 sec_num = atoi (*argv + 2); break;
66:
67:             default:
68:                 printf ("オプション・エラーです．:¥"%c¥"¥n", argv[0][1]);
69:                 exit (1);
```

番号を AX レジスタに返します．

サブルーチン func_36 は，引数としてドライブ番号と構造体 _DSK へのポインタを受け取ります．そして，ファンクション 36H を用いて指定されたドライブのディスク情報を読み出し，ポインタで指定された構造体の各メンバにそのディスク情報を格納します．

サブルーチン int_25 は，引数としてドライブ番号，バッファへのポインタ，開始セクタ，セクタ数を受け

〔リスト5-7〕プログラム ddump.c ②

```
 70:                     break;
 71:             }
 72:         } else {
 73:             drv_name = *argv;
 74:         }
 75:     }
 76:     if (drv_name == NULL) {
 77:         drv_num = func_19 ();
 78:     } else {
 79:         drv_num = toupper(*drv_name) - 'A';
 80:     }
 81:     ddump (drv_num, sec_begin, sec_num);
 82:     printf ("¥n");
 83:     exit (0);
 84: }
 85:
 86: /*****************************************************************
 87: *
 88: *     関数名 :   ddump (drv_num, sec_begin, sec_num)
 89: *     機  能 :   ディスク・ダンプ
 90: *     入  力 :   drv_num    ‥‥  ドライブ番号
 91: *                sec_begin  ‥‥  開始セクタ
 92: *                sec_num    ‥‥  ダンプするセクタ数
 93: *     出  力 :   なし
 94: *
 95: *****************************************************************/
 96: void ddump (drv_num, sec_begin, sec_num)
 97: int drv_num, sec_begin, sec_num;
 98: {
 99:     DSK disk_data;
100:     char *buff;
101:     int st;
102:
103:     func_36 (drv_num + 1, &disk_data);
104:     if (disk_data.ax == 0xFFFF) {
105:         printf ("ドライブの指定が無効です.¥n");
106:         exit (2);
107:     }
108:     disk_msg (drv_num, &disk_data);
109:     if ((buff = calloc (1, disk_data.ax)) == NULL) {
110:         printf ("セクタ・バッファが確保できません.¥n");
111:         exit (3);
112:     }
113:     while (sec_num--) {
114:         if ((st = int_25 (drv_num, buff, 1, sec_begin))) {
115:             printf ("セクタの読み出し中にエラーが発生しました.¥n");
116:             printf ("エラー・コード ----- %d¥n", st);
117:             exit (4);
118:         }
119:         printf ("¥n論理セクタ番号 = %d¥n", sec_begin);
120:         dump (buff, disk_data.cx);
121:         sec_begin++;
122:     }
123: }
124: /*****************************************************************
125: *
126: *     関数名 :   disk_msg (drv_num, ptr)
127: *     機  能 :   ディスク情報の表示
128: *     入  力 :   drv_num   ‥‥ ドライブ番号
129: *                ptr        ‥‥ ディスク・データ構造体へのポインタ
130: *     出  力 :   なし
131: *
132: *****************************************************************/
133: void disk_msg (drv_num, ptr)
134: DSK *ptr;
135: int drv_num;
136: {
137:     long bytes0, bytes1;
138:
```

取り，INT 25H の内部割り込みを用いて指定セクタの読み出しを行います．読み出されたセクタ・データは，引数で指定されたバッファに格納されます．ここでエラーがなければ AX レジスタを 0 にして返し，エラーが発生した場合は，INT 25H から返されたエラー・コードをそのまま AX レジスタに返します．

◆ **生成方法**

プログラム ddump.exe は，次の手順で分割アセンブ

〔リスト5-7〕プログラム ddump.c ③

```
139:     bytes1 = (long)ptr -> bx * ptr -> ax * ptr -> cx;
140:     bytes0 = (long)ptr -> dx * ptr -> ax * ptr -> cx;
141:     printf ("¥nドライブ %c のディスク情報¥n¥n", (char)(drv_num + 'A'));
142:     printf ("1ドライブあたりのクラスタ数    ---   %d クラスタ¥n", ptr -> dx);
143:     printf ("1クラスタあたりのセクタ数      ---   %d セクタ¥n", ptr -> ax);
144:     printf ("1セクタあたりのバイト数        ---   %d バイト¥n", ptr -> cx);
145:     printf ("ディスク容量                   ---   %ld バイト¥n", bytes0);
146:     printf ("残り容量                       ---   %ld バイト¥n", bytes1);
147: }
148:
149: /*****************************************************************
150:  *
151:  *    関数名 :    dump (buff, byte)
152:  *    機  能 :    メモリ・ダンプ ( 256バイト単位 )
153:  *    入  力 :    buff ･･･ データ領域へのポインタ
154:  *                byte ･･･ バイト数
155:  *    出  力 :    なし
156:  *
157:  *****************************************************************/
158: void dump (buff, byte)
159: char *buff;
160: unsigned int byte;
161: {
162:     char *i;
163:     long count = 0;
164:
165:     do {
166:         printf ("¥n");
167:         for (i = buff; i < buff + 0x100; i += 16) {
168:             para_dump (count, i);
169:             count += 16;
170:         }
171:         buff += 0x100;
172:     } while (byte -= 0x100);
173: }
174:
175: /*****************************************************************
176:  *
177:  *    関数名 :    para_dump (count, buff)
178:  *    機  能 :    メモリ・ダンプ ( 16バイト単位 )
179:  *    入  力 :    count ･･･ データの先頭からのオフセット
180:  *                buff  ･･･ データ領域へのポインタ
181:  *    出  力 :    なし
182:  *
183:  *****************************************************************/
184: void para_dump (count, buff)
185: long count;
186: char *buff;
187: {
188:     char *i;
189:     char c;
190:
191:     printf (" %061X : ", count);
192:     for (i = buff; i < buff + 8; i++) {
193:         printf ("%02X ", *i & 0x00FF);
194:     }
195:     printf ("- ");
196:     for (;i < buff + 16; i++) {
197:         printf ("%02X ", *i & 0x00FF);
198:     }
199:     printf ("  ");
200:     for (i = buff; i < buff + 16; i++) {
201:         c = *i;
202:         if (c < ' ' || c >= 0x7F) {
203:             c = '.';
204:         }
205:         printf ("%c", c);
206:     }
207:     printf ("¥n");
208: }
```

ル/コンパイルして作成します.

masm /ML ddumpsub;

cl −J −As ddump.c ddumpsub

◆ **実行サンプル**

リスト5-9 はプログラム ddump.exe の実行例を示しています.

① まず,ドライブBのルート・ディレクトリを2セクタ分ダンプしてみる.ドライブBは1Mバイト・フロッピ・ディスクである旨の表示がされる.

② 1Mバイト・フロッピ・ディスクの場合,ルート・ディレクトリは5セクタから始まる.

③ 次のセクタ(2セクタ目)も表示される.

④ 次にASSIGNされたハード・ディスク(ドライブH)のルート・ディレクトリをダンプしてみる.

〔**リスト5-8**〕
プログラム
ddumpsub.asm ①

```
 1: ;*********************************************************************
 2: ;
 3: ;   機     能 :    ディスク・ダンプ・サブルーチン
 4: ;   割り込み :    INT 25H(ディスクの直接読み出し)
 5: ;   ファンクション :    19H(カレント・ドライブ番号の読み出し)
 6: ;                   36H(ディスク・データの読み出し)
 7: ;   生     成 :    masm /ML ddumpsub;
 8: ;
 9: ;*********************************************************************
10:             .MODEL  SMALL, C
11:             .CODE
12: ;*********************************************************************
13: ;
14: ;   構造体 :    _DSK
15: ;   機  能 :    ディスク・データ構造の定義
16: ;
17: ;*********************************************************************/
18: _DSK        STRUC
19: ax_data     DW      ?
20: bx_data     DW      ?
21: cx_data     DW      ?
22: dx_data     DW      ?
23: _DSK        ENDS
24:
25: ;*********************************************************************
26: ;
27: ;   ルーチン名 :  func_19
28: ;   機    能 :  カレント・ドライブ番号の取得
29: ;   func    :  19H(カレント・ドライブ番号の読み出し)
30: ;   入    力 :  なし
31: ;   出    力 :  AX ・・ドライブ番号 ( 00H=A, 01H=B, ・・)
32: ;
33: ;*********************************************************************
34: func_19     PROC
35:             mov     ah, 19h         ;ファンクション19H
36:             int     21h
37:             xor     ah, ah
38:             ret
39: func_19     ENDP
40:
41: ;*********************************************************************
42: ;
43: ;   ルーチン名 :  func_36
44: ;   機    能 :  ディスク・データの読み出し
45: ;   func    :  36H(ディスク残り容量の読み出し)
46: ;   入    力 :  arg1  ・ドライブ番号
47: ;                arg2  ・データ構造体へのポインタ
48: ;   出    力 :  なし(構造体の中へ設定)
49: ;
50: ;*********************************************************************
51: func_36     PROC    arg1:WORD, arg2:PTR
52:             push    si
53:             mov     dx, arg1        ;ドライブ番号
54:             mov     ah, 36h         ;ファンクション36H
55:             int     21h
56:             mov     si, arg2
57:             mov     [si.ax_data], ax    ;1クラスタあたりのセクタ数
58:             mov     [si.bx_data], bx    ;使用可能なクラスタ数
59:             mov     [si.cx_data], cx    ;1セクタあたりのバイト数
60:             mov     [si.dx_data], dx    ;1ドライブあたりのクラスタ数
61:             pop     si
62:             ret
63: func_36     ENDP
64:
```

〔リスト5-8〕
プログラム
ddumpsub.asm ②

```
65: ;*********************************************************************
66: ;
67: ;     ルーチン名 :   int_25
68: ;     機    能 :   セクタ・データの読み出し
69: ;     割    込 :   INT 25H(ディスクの直接読み出し)
70: ;     入    力 :   arg1 ··· ドライブ番号
71: ;                  arg2 ··· バッファへのポインタ
72: ;                  arg3 ··· セクタ数
73: ;                  arg4 ··· セクタ番号
74: ;     出    力 :   AX ··· エラー・コード(0:なし)
75: ;
76: ;*********************************************************************
77: int_25          PROC    arg1:WORD, arg2:PTR, arg3:WORD, arg4:WORD
78:                 push    si
79:                 push    di
80:                 mov     ax, arg1                ;ドライブ番号
81:                 mov     bx, arg2                ;バッファへのポインタ
82:                 mov     cx, arg3                ;セクタ数
83:                 mov     dx, arg4                ;セクタ番号
84:                 int     25h                     ;INT 25H
85:                 pop     dx
86:                 jb      err
87:                 xor     al, al
88: err:
89:                 xor     ah, ah
90:                 pop     di
91:                 pop     si
92:                 ret
93: int_25          ENDP
94:                 END
```

〔リスト5-9〕**プログラム ddump.exe の実行例 ①**

```
R>ddump b: -s5 -n2 ⏎··· ドライブBのルート・ディレクトリ(セクタ5)をダンプ①

 *** ディスク・ダンプ・プログラム Ver.1.1 ***

ドライブ B のディスク情報

1ドライブあたりのクラスタ数      ---   1221 クラスタ   ┐
1クラスタあたりのセクタ数        ---   1 セクタ        │
1セクタあたりのバイト数          ---   1024 バイト     ├1Mバイト・フォーマット・ディスク
ディスク容量                     ---   1250304 バイト  │
残り容量                         ---   27648 バイト    ┘

論理セクタ番号 = 5 ···ディレクトリ・セクタ②

 000000 : 49 4F 20 20 20 20 20 20 - 53 59 53 27 00 00 00 00    IO      SYS'....
 000010 : 00 00 00 00 00 00 02 00 - ED 10 02 00 00 00 01 00    ................
 000020 : 4D 53 44 4F 53 20 20 20 - 53 59 53 27 00 00 00 00    MSDOS   SYS'....
 000030 : 00 00 00 00 00 00 02 00 - ED 10 42 00 40 72 00 00    ..........B.@r..
 000040 : 43 4F 4D 4D 41 4E 44 20 - 43 4F 4D 20 00 00 00 00    COMMAND COM ....
 000050 : 00 00 00 00 00 00 02 00 - ED 10 5F 00 63 61 00 00    ..........._.ca..
 000060 : 41 44 44 44 52 56 20 20 - 45 58 45 20 00 00 00 00    ADDDRV  EXE ....
 000070 : 00 00 00 00 00 00 02 00 - ED 10 78 00 D0 47 00 00    ..........x..G..
 000080 : 41 53 53 49 47 4E 20 20 - 45 58 45 20 00 00 00 00    ASSIGN  EXE ....
 000090 : 00 00 00 00 00 00 02 00 - ED 10 8A 00 52 67 00 00    ............Rg..
 0000A0 : 41 54 54 52 49 42 20 20 - 45 58 45 20 00 00 00 00    ATTRIB  EXE ....
 0000B0 : 00 00 00 00 00 00 02 00 - ED 10 A4 00 C2 23 00 00    ............#..
 0000C0 : 41 50 50 45 4E 44 20 20 - 43 4F 4D 20 00 00 00 00    APPEND  COM ....
 0000D0 : 00 00 00 00 00 00 02 00 - ED 10 AD 00 04 08 00 00    ................
```

⑤ ASSIGN されたドライブ名が表示される．同様に20 M バイト・フォーマットのハード・ディスクである旨の表示もされる．

⑥ ハード・ディスク(20 M バイト)の場合，ルート・ディレクトリは9セクタ目から始まる．

⑦ カレント・ドライブ(RAM ディスク)のルート・ディレクトリをダンプしてみる．

⑧ ASSIGN された RAM ディスクのドライブ名(R)が表示される．

⑨ カレント・ドライブは，1.5 M バイトの RAM ディスクであることがわかる．

⑩ RAM ディスクのルート・ディレクトリは4セクタ目から始まっている．

〔リスト5-9〕プログラム ddump.exe の実行例 ②

```
      0000E0 : 42 41 43 4B 55 50 20 20 - 45 58 45 20 00 00 00 00    BACKUP  EXE ....
      0000F0 : 00 00 00 00 00 00 02 00 - ED 10 B0 00 82 62 00 00    ............b..

      000100 : 43 48 4B 44 53 4B 20 20 - 45 58 45 20 00 00 00 00    CHKDSK  EXE ....
      000110 : 00 00 00 00 00 00 02 00 - ED 10 C9 00 90 28 00 00    ..............(..
      000120 : 43 55 53 54 4F 4D 20 20 - 45 58 45 20 00 00 00 00    CUSTOM  EXE ....
                               :
                               :
      0003E0 : 47 52 41 50 48 20 20 20  - 4C 49 42 20 00 00 00 00    GRAPH   LIB ....
      0003F0 : 00 00 00 00 00 00 02 00  - ED 10 E3 02 00 40 02 00    .............@..

     論理セクタ番号 = 6 … 次のセクタ③

      000000 : 47 52 41 50 48 20 20 20 - 53 59 53 20 00 00 00 00    GRAPH   SYS ....
      000010 : 00 00 00 00 00 00 02 00 - ED 10 73 03 6F 87 00 00    ...........s.o..
      000020 : 4D 4F 55 53 45 20 20 20 - 53 59 53 20 00 00 00 00    MOUSE   SYS ....
      000030 : 00 00 00 00 00 00 02 00 - ED 10 95 03 91 0F 00 00    ................
                               :
                               :
      0003D0 : E5 E5 E5 E5 E5 E5 E5 E5 - E5 E5 E5 E5 E5 E5 E5 E5    ................
      0003E0 : 00 E5 E5 E5 E5 E5 E5 E5 - E5 E5 E5 E5 E5 E5 E5 E5    ................
      0003F0 : E5 E5 E5 E5 E5 E5 E5 E5 - E5 E5 E5 E5 E5 E5 E5 E5    ................

     R>ddump h: -s9 -n1 ⏎… ASSIGNされたハード·ディスクのルート·ディレクトリをダンプ④

      *** ディスク・ダンプ・プログラム Ver.1.1 ***

     ドライブ H のディスク情報   (ASSIGNされたドライブ名⑤)

     1ドライブあたりのクラスタ数   ---   2468 クラスタ
     1クラスタあたりのセクタ数     ---   8 セクタ
     1セクタあたりのバイト数       ---   1024 バイト        } 20Mバイト·ハード·ディスク
     ディスク容量                  ---   20217856 バイト
     残り容量                      ---   7094272 バイト

     論理セクタ番号 = 9 … ルート·ディレクトリ·セクタ⑥

      000000 : 49 4F 20 20 20 20 20 20 - 53 59 53 27 00 00 00 00    IO      SYS'....
      000010 : 00 00 00 00 00 00 01 00 - 57 0F 02 00 00 C0 00 00    ........W.......
      000020 : 4D 53 44 4F 53 20 20 20 - 53 59 53 27 00 00 00 00    MSDOS   SYS'....
                               :
                               :
      0003D0 : 00 00 00 00 00 00 75 6B - 71 11 D6 03 DC 19 00 00    ......ukq.......
      0003E0 : E5 45 4E 20 20 20 20 20 - 43 54 4C 20 00 00 00 00    .EN     CTL ....
      0003F0 : 00 00 00 00 00 00 75 6B - 71 11 8C 05 04 07 00 00    ......ukq.......

     R>ddump -s4 ⏎… カレント·ドライブのルート·ディレクトリをダンプ⑦

      *** ディスク・ダンプ・プログラム Ver.1.1 ***

     ドライブ R のディスク情報   (ASSIGNされたRAMディスク⑧)

     1ドライブあたりのクラスタ数   ---   760 クラスタ
     1クラスタあたりのセクタ数     ---   4 セクタ
     1セクタあたりのバイト数       ---   512 バイト         } 1.5MバイトRAMディスク⑨
     ディスク容量                  ---   1556480 バイト
     残り容量                      ---   102400 バイト

     論理セクタ番号 = 4 … ルート·ディレクトリ·セクタ⑩

      000000 : 5B 49 4F 53 2D 31 30 20 - 20 20 5D 08 00 00 00 00    [IOS-10   ]....
      000010 : 00 00 00 00 00 00 2F 41 - 81 11 00 00 00 00 00 00    ....../A........
      000020 : 43 4F 4D 4D 41 4E 44 20 - 43 4F 4D 20 00 00 00 00    COMMAND COM ....
                               :
                               :
      0001D0 : 00 00 00 00 00 00 01 00 - 3D 11 25 02 D5 96 00 00    ........=.%.....
      0001E0 : 55 50 20 20 20 20 20 20 - 43 20 20 20 00 00 00 00    UP      C   ....
      0001F0 : 00 00 00 00 00 00 1A 73 - 47 0B 44 02 95 02 00 00    .......sG.D.....

     R>
```

Absolute Disk Write		INT 26H
機能	ディスク・セクタへの直接書き込み	
コール	AL←ドライブ番号 (00H=A, 01H=B, …) DS:BX←ディスク転送アドレス CX←書き込みたいセクタ数 DX←書き込み開始セクタ	
リターン	CF=1 エラー発生 AL レジスタにエラー・コードを返す CF=0 正常終了	

この内部割り込みの使いかたや処理内容については，データが読み込まれるのではなく，書き込まれるということの相違を除けば INT 25H とまったく同様です．

Terminate But Stay Resident		INT 27H
機能	プログラムをメモリに常駐したまま終了	
コール	CS:DX←常駐させるプログラムの最終アドレスの次のアドレス	
リターン	なし	

この割り込みも INT 20H と同様に，CS レジスタが PSP を指していなければならないため COM モデルにのみ適用されます．

EXE モデルの場合は，ファンクション 31H(159 ページ)によるシステム・コールを使用すべきです．また，この割り込みにより実行可能な COM ファイルがメモリに常駐すると，以後そのコマンドは内部コマンドとして使用可能になります．

Write to Special Device		INT 29H
機能	特殊デバイスに対する文字の出力	
コール	AL←文字コード	
リターン	なし	

この内部割り込みでは，特殊デバイスである CON(スクリーン)に対して文字の出力を行います．このシステム・コールによる文字の出力では，このあとで述べるファンクション・リクエストと比較して，制御文字のチェックを行わないため，ファンクション・リクエストよりも高速な処理が可能となります．

ただし，このシステム・コールは将来のバージョンでは使用できなくなる可能性があります．

* *

ここでは，MS-DOS がユーザに解放している割り込みについて解説しました．これらの割り込みのうち，INT 20H や INT 27H は，次章のファンクション・リクエストで代用できるのと，EXE モデルでは利用しにくいため，あまり使用することはないでしょうし，使用しないほうが無難です．

INT 22H～INT 24H は，単なるアドレス格納領域です．中断アドレスや致命的エラーは，デフォルトで command.com 内の処理ルーチンが利用できることになります．しかし，ある種のアプリケーションでは独自の中断処理を行ったり，独自のエラー・リカバリを必要とする場合が生じてきます．したがって，これらの処理アドレスに関する知識は，PSP 内の該当するフィールドとともに，少し複雑(高級？)なアプリケーション・プログラムを作成する場合には必須のものとな

● 8086 vs 68000(その4) メモリ空間 ●

CP/M が幅をきかせていた時代の 8 ビット CPU では，そのメモリ空間が 2^{16}，すなわち 64 K バイトでした．それが，8086 になって 2^{20}=1 M バイトに拡大したため，日本語処理やグラフィックスの処理において歴然とした能力の差が出てしまいました．

8086 と 8 ビット CPU では，CPU 自身の処理能力にもそれなりの差があることは否めません．しかし，この 8 ビット/16 ビットの差は，なんといってもメモリ容量の差ではないでしょうか．8 ビット CPU に比較して 8086 では，2^4=16 倍もの情報量を扱うことができるのです．

そして 68000 に至ると，最大 64 M バイトのメモリ空間をアクセスすることが可能になります．なんと 8086 の 64 倍ではありませんか．さらに，68000 には 8086 にみられるセグメントなどという「厄介者」は一切ありません．自動車にたとえるならば，凸凹した砂利道を走るのと舗装された高速道路を走るほどの差があるといっても過言ではないでしょう．

ります．

INT 25HやINT 26Hによるディスクへの直接アクセスは，ディレクトリの解析を行ったりする特殊なアプリケーション・プログラムには必要となるシステム・コールです．これらの割り込みは注意して使用しないとディスクの破壊にもつながりかねません．しかし，これらの割り込みを利用して，たとえばディレクトリのソート・プログラムなどに挑戦してみるのもおもしろいでしょう．

INT 29Hは，一般的には公開されていません．したがって本文でも述べたように，この割り込みは将来的にも利用できるという保証はありません．しかし，この割り込みはファンクション・リクエストと競合しないために，第7章のアプリケーションが示すようにファンクション・リクエストが利用できない場合に重宝します．

基本的に，あまり行儀の良くないプログラミングは避けたほうが良いのですが….

第6章 MS-DOSのファンクション・リクエスト

ファンクションとパラメータとリターン

MS-DOSでは，その機能の一部をシステム・コールの形でユーザに解放しています．システム・コールのなかでも汎用性をもっているものは，ファンクション・リクエストとしてサービスされています．ここでは，MS-DOS ver.3.30のファンクション・リクエストの機能と使いかたについて，機能タイプ別に分類して整理しておきます．

表6-1は，ver.3.30におけるファンクション・リクエストの一覧です．

MS-DOSは，開発当初(ver.1.25)においてはCP/Mの発展型として登場し，ver.2.11以降になってUNIXの概念を導入したという経緯をもっています．したがって，システム・コール(ファンクション・リクエスト)においても，ver.1.25時代のCP/Mコンパチブルなファンクション・リクエストと，ver.2.11以降のUNIXを指向したファンクション・リクエストの両者をサポートしています．

ver.3.30では，ファンクション0～62Hまでの84種類のファンクション・リクエストが用意されています．いくつかのファンクションでは，さらにALレジスタを用いて細かい機能の指定を行えるサブファンクションも用意されています．

表6-1のファンクションのうち00H～24Hのファンクションは，ver.1.25レベルのファンクション・リクエストであり，ほぼCP/Mとの互換性が保たれています．これらのファンクションでは，ファイルのアクセスに関してはFCBを使用し，エラーの有無をALレジスタに返します．

〔表6-1〕ファンクション・リクエスト一覧 ①

番号	ファンクション名	機能	コール	リターン
00H	Terminate Program	プログラムの終了	AH=00H CS←PSPのセグメント・アドレス	なし
01H	Read Keyboard and Echo	キーボード入力とエコー	AH=01H	AL←入力された文字コード
02H	Display Character	文字のスクリーン出力	AH=02H DL←出力すべき文字コード	なし
03H	Auxiliary Input	補助入力	AH=03H	AL←補助装置から入力された文字コード
04H	Auxiliary Output	補助出力	AH=04H DL←出力すべき文字コード	なし
05H	Print Character	プリンタ出力	AH=05H DL←出力すべき文字コード	なし
06H	Direct Console I/O	直接コンソール入出力	AH=06H DL≠FFH：出力文字コード DL=FFH：文字入力	入力の場合：AL←入力文字コード (ZF=0) AL=00H(ZF=1)
07H	Direct Console Input	直接コンソール入力	AH=07H	AL←入力された文字コード
08H	Read Keyboard	キーボード入力	AH=08H	AL←入力された文字コード
09H	Display String	文字列のスクリーンへの出力	AH=09H DS：DX←文字列の先頭アドレス	なし
0AH	Buffered Keyboard Input	バッファード・キーボード入力	AH=0AH DS：DX←バッファへのポインタ	なし
0BH	Check Keyboard Status	キーボード・バッファのチェック	AH=0BH	AL=FFH：バッファに文字が入っている AL=00H：バッファに文字が入っていない

〔表6-1〕ファンクション・リクエスト一覧 ②

番号	ファンクション名	機　能	コ ー ル	リ タ ー ン
0CH	Flush Buffer, Read Keyboard	バッファを空にしてキーボード入力	AH=0CH AL←ファンクション・コード (01H, 06H, 07H, 08H, 0AH) AL←それ以外：バッファを空にする	AL=00H：バッファが空になっている
0DH	Reset Disk	ディスクのリセット	AH=0DH	なし
0EH	Select Disk	ディスクの選択	AH=0EH DL←ドライブ番号 (00H=A, 01H=B, …)	AL←論理ドライブ数
0FH	Open File	ファイルのオープン	AH=0FH DS：DX←オープンされていないFCB	AL=00H：正常終了 AL=FFH：ディレクトリ・エントリが見つからない
10H	Close File	ファイルのクローズ	AH=10H DS：DX←オープンされているFCB	AL=00H：正常終了 AL=FFH：ディレクトリ・エントリが存在しない
11H	Search for First Entry	最初に一致するエントリの検索	AH=11H DS：DX←オープンされていないFCB	AL=00H：正常終了 AL：FFH：ディレクトリ・エントリが存在しない
12H	Search for Next Entry	次に一致するエントリの検索	AH=12H DS：DX←オープンされていないFCB	AL=00H：正常終了 AL=FFH：ディレクトリ・エントリが存在しない
13H	Delete File	ファイルの削除	AH=13H DS：DX←オープンされていないFCB	AL=00H：正常終了 AL=FFH：ディレクトリ・エントリが存在しない
14H	Sequential Read	シーケンシャルな読み込み	AH=14H DS：DX←オープンされているFCB	AL=00H：正常終了 AL=01H：EOF検出 AL=02H：DTAが小さい AL=03H：レコードの一部分の読み込み
15H	Sequential Write	シーケンシャルな書き込み	AH=15H DS：DX←オープンされているFCB	AL=00H：正常終了 AL=01H：ディスクに空き領域がない AL=02H：DTAが小さい
16H	Create File	ファイルの作成	AH=16H DS：DX←オープンされていないFCB	AL=00H：正常終了 AL=FFH：空のディレクトリ・エントリが存在しない
17H	Rename File	ファイル名の変更	AH=17H DS：DX←変更すべきファイル名の入ったFCB	AL=00H：正常終了 AL=FFH：目的のディレクトリ・エントリが存在しないか，ファイル名がすでに存在する
19H	Get Current Disk	カレント・ドライブ番号の取得	AH=19H	AL←ドライブ番号 (00H=A, 01H=B, …)
1AH	Set Disk Transfer Address	ディスク転送アドレスのセット	AH=1AH DS：DX←ディスク転送アドレス	なし
1BH	Get Default Drive Data	デフォルト・ドライブのデータ取得	AH=1BH	AL←1クラスタ当たりのセクタ数 CX←1セクタ当たりのバイト数 DX←1ドライブ当たりのクラスタ数 DS：BX←FAT-IDへのポインタ
1CH	Get Drive Data	指定ドライブのデータ取得	AH=1CH DL←ドライブ番号 (00H=カレント, 01H=A, …)	AL=FFH：ドライブ番号の指定が無効 FFH以外：1クラスタ当たりのセクタ数 CX←1セクタ当たりのバイト数 DX←1ドライブ当たりのクラスタ数 DS：BX←FAT-IDへのポインタ
21H	Random Read	ランダムな読み出し	AH=21H DS：DX←オープンされているFCB	AL=00H：正常終了 AL=01H：EOFの検出 AL=02H：DTAが小さい AL=03H：レコードの一部分の読み込み

〔表6-1〕ファンクション・リクエスト一覧 ③

番号	ファンクション名	機能	コール	リターン
22H	Random Write	ランダムな書き込み	AH=22H DS：DX←オープンされているFCB	AL=00H：正常終了 AL=01H：ディスクに空き領域がない AL=02H：DTAが小さい
23H	Get File Size	ファイル・サイズの取得	AH=23H DS：DX←オープンされていないFCB	AL=00H：正常終了 AL=FFH：ディレクトリ・エントリが存在しない
24H	Set Relative Record	相対レコードの設定	AH=24H DS：DX←オープンされていないFCB	なし
25H	Set Interrupt Vector	割り込みベクタの設定	AH=25H AL←割り込みタイプ番号 DS：DX←割り込み処理ルーチン・アドレス	なし
26H	Create New PSP	新しいPSPの作成	AH=26H DX←新しいPSPのセグメント・アドレス	なし
27H	Random Block Read	ランダムなブロック読み出し	AH=27H DS：DX←オープンされているFCB CX←読み出すべきレコード数	AL=00H：正常終了 AL=01H：EOFの検出 AL=02H：DTAが小さい AL=03H：レコードの一部分の読み込み CX←読み出されたレコード数
28H	Random Block Write	ランダムなブロック書き込み	AH=28H DS：DX←オープンされているFCB CX←書き込むべきレコード数 （0：ファイル・サイズ・フィールドの設定）	AL=00H：正常終了 AL=01H：ディスクの空き領域がない AL=02H：DTAが小さい CX←書き込まれたレコード数
29H	Parse File Name	ファイル名の解析	AH=29H AL←解析の制御 DS：SI←解析すべき文字列へのポインタ ES：DI←オープンされていないFCB	AL=00H：ワイルド・カードは使用されていない AL=01H：ワイルド・カードが使用されている AL=FFH：ドライブ名が無効 DS：SI←解析された文字列の中のファイル名の直後のアドレス
2AH	Get Date	日付の読み出し	AH=2AH	CX←年(1980～2099) DH←月(1～12) DL←日(1～31) AL←曜日(0=日，1=月，…6=土)
2BH	Set Date	日付の設定	AH=2BH CX←年(1980～2099) DH←月(1～12) DL←日(1～31)	AL=00H：正常終了 AL=FFH：無効な日付
2CH	Get Time	時刻の読み出し	AH=2CH	CH←時(0～23) CL←分(0～59) DH←秒(0～59) DL←1/100秒(0～99)
2DH	Set Time	時刻の設定	AH=2DH CH←時(0～23) CL←分(0～59) DH←秒(0～59) DL←1/100秒(0～99)	AL=00H：正常終了 AL=FFH：無効な時刻
2EH	Set/Reset Verify Flag	ベリファイ・フラグの設定	AH=2EH AL=00H：ベリファイを行わない AL=01H：ベリファイを行う	なし
2FH	Get Disk Transfer Address	ディスク転送アドレスの読み出し	AH=2FH	ES：BX←ディスク転送アドレス
30H	Get MS-DOS Version Number	DOSのバージョン番号の読み出し	AH=30H	AL←バージョン番号の整数部 AH←バージョン番号の小数部 BH←OEMのシリアル番号 BL：CX←24ビットのユーザ番号

〔表6-1〕ファンクション・リクエスト一覧 ④

番号	ファンクション名	機　能	コ ー ル	リ タ ー ン
31H	Keep Process	プログラムの常駐終了	AH＝31H AL←終了コード DX←パラグラフ(16バイト単位)でのメモリ・サイズ	なし
33H	ctrl-C Check	ブレーク・チェック・フラグの読み出し/設定	AH＝33H AL＝00H：フラグの読み出し AL＝01H：フラグの設定 DL＝00H：OFF DL＝01H：ON	AL＝FFH：フラグ設定が無効 DL＝00H：OFF DL＝01H：ON
35H	Get Interrupt Vector	割り込みベクタの読み出し	AH＝35H AL←割り込み番号	ES：BX←割り込みルーチン・アドレス
36H	Get Disk Free Space	ディスク残り領域の読み出し	AH＝36H DL←ドライブ番号 (00H＝カレント，01H＝A，…)	BX←使用可能なクラスタ数 DX←1ドライブ当たりのクラスタ数 CX←1セクタ当たりのバイト数 AX←1クラスタ当たりのセクタ数 AX＝FFFFH：ドライブ番号が無効
38H	Get Country Data	国別情報の読み出し	AH＝38H AL＝00H：現在の国 AL＝01H：USA 規格 AL＝51H：日本規格 DS：DX←バッファ(32バイト)へのポインタ	CF＝0：正常終了 CF＝1：AX←エラー・コード
38H	Set Country Data	国別情報の設定	AH＝38H DX＝FFFFH AL←国別コード(FFH 以外) BH←FFH 以上の国別コード (AL＝FFH)	CF＝0：正常終了 CF＝1：AX←エラー・コード
39H	Create Directory	ディレクトリの作成	AH＝39H DS：DX←パス名の先頭アドレス	CF＝0：正常終了 CF＝1 AX←エラーコード
3AH	Remove Directory	ディレクトリの削除	AH＝3AH DS：DX←パス名の先頭アドレス	CF＝0：正常終了 CF＝1：AX←エラー・コード
3BH	Change Current Directory	カレント・ディレクトリの変更	AH＝3BH DS：DX←パス名の先頭アドレス	CF＝0：正常終了 CF＝1：AX←エラー・コード
3CH	Create Handle	ハンドルの作成	AH＝3CH DS：DX←パス名の先頭アドレス CX←ファイルの属性	CF＝0：AX←ファイル・ハンドル CF＝1：AX←エラー・コード
3DH	Open Handle	ハンドルのオープン	AH＝3DH AL←アクセス制御 DS：DX←パス名の先頭アドレス	CF＝0：AX←ファイル・ハンドル CF＝1：AX←エラー・コード
3EH	Close Handle	ハンドルのクローズ	AH＝3EH BX←ファイル・ハンドル	CF＝0 正常終了 CF＝1：AX←エラー・コード
3FH	Read Handle	ハンドルの読み出し	AH＝3FH DS：DX←バッファへのポインタ CX←読み込むバイト数 BX←ファイル・ハンドル	CF＝0：AX←読み込まれたバイト数 CF＝1：AX←エラー・コード
40H	Write Handle	ハンドルへの書き込み	AH＝40H DS：DX←バッファへのポインタ CX←書き込むバイト数 BX←ファイル・ハンドル	CF＝0：AX←書き込まれたバイト数 CF＝1：AX←エラー・コード
41H	Delete Directory Entry	ファイルの削除	AH＝41H DS：DX←パス名の先頭アドレス	CF＝0：正常終了 CF＝1：AX←エラー・コード
42H	Move File Pointer	ファイル・ポインタの移動	AH＝42H CX：DX←移動するバイト数(符号つき32ビット) AL←移動方法 BX←ファイル・ハンドル	CF＝0：DX：AX←移動後のファイル・ポインタ CF＝1：AX←エラー・コード

〔表6-1〕ファンクション・リクエスト一覧 ⑤

番号	ファンクション名	機　能	コ　ー　ル	リ　タ　ー　ン
43H	Get/Set File Attributes	ファイル属性の取得/変更	AH＝43H DS：DX ← パス名の先頭アドレス AL ← ファンクション (00H：読み出し，01H：変更) CX ← 設定すべき属性	CF＝0：CX ← 属性 CF＝1：AX ← エラー・コード
4400H	Get IOCTL Data	IOCTL データの読み出し	AX＝4400H BX ← ハンドル	CF＝0：DX ← デバイス・データ CF＝1：AX ← エラー・コード
4401H	Set IOCTL Data	IOCTL データの設定	AX＝4401H BX ← ハンドル DX ← デバイス・データ(DH＝00H)	CF＝0：DX ← デバイス・データ CF＝1：AX ← エラー・コード
4402H	Receive IOCTL Character	IOCTL(キャラクタ)の取得	AX＝4402H BX ← ハンドル CX ← IOCTL データのバイト数 DS：DX ← バッファへのポインタ	CF＝0：AX ← 転送されたバイト数 CF＝1：AX ← エラー・コード
4403H	Send IOCTL Character	IOCTL(キャラクタ)の送出	AX＝4403H BX ← ハンドル CX ← IOCTL データのバイト数 DS：DX ← バッファへのポインタ	CF＝0：AX ← 転送されたバイト数 CF＝1：AX ← エラー・コード
4404H	Receive IOCTL Block	IOCTL(ブロック)の取得	AX＝4404H BL ← ドライブ番号 (00H＝デフォルト，01H＝A，…) CX ← IOCTL データのバイト数 DS：DX ← バッファへのポインタ	CF＝0：AX ← 転送されたバイト数 CF＝1：AX ← エラー・コード
4405H	Send IOCTL Block	IOCTL(ブロック)の送出	AX＝4405H BL ← ドライブ番号 (00H＝デフォルト，01H＝A，…) CX ← IOCTL データのバイト数 DS：DX ← バッファへのポインタ	CF＝0：AX ← 転送されたバイト数 CF＝1：AX ← エラー・コード
4406H	Get Input IOCTL Status	入力ステータスのチェック	AX＝4406H BX ← ハンドル	CF＝0：AL＝00H：レディ状態でない AL＝FFH：レディ状態 CF＝1：AX ← エラー・コード
4407H	Get Output IOCTL Status	出力ステータスのチェック	AX＝4407H BX ← ハンドル	CF＝0：AL＝00H：レディ状態でない AL＝FFH：レディ状態 CF＝1：AX ← エラー・コード
4408H	IOCTL is Changeable	IOCTL の交換性	AX＝4408H BL ← ドライブ番号 (00H＝デフォルト，01H＝A，…)	CF＝0：AX＝00H：交換可能 AX＝01H：交換不可能 CF＝1：AX ← エラー・コード
4409H	IOCTL is Redirected Block	IOCTL リダイレクト(ブロック)	AX＝4409H BL＝ドライブ番号 (00H＝デフォルト，01H＝A，…)	CF＝0：DX ← デバイス属性 CF＝1：AX ← エラー・コード
440AH	IOCTL is Redirected Handle	IOCOL リダイレクト(ハンドル)	AX＝440AH BX ← ハンドル	CF＝0：DX ← IOCTL ビット・フィールド CF＝1：AX ← エラー・コード
440BH	IOCTL Retry	リトライ回数の設定	AX＝440BH DX ← リトライの回数 CX ← 待ち時間	CF＝0：正常終了 CF＝1：AX ← エラー・コード
440CH	Generic IOCTL (for handles)	一般 IOCTL(ハンドル用)	AX＝440CH BX ← ハンドル CH＝05H：カテゴリ・コード (プリンタ・デバイス) CL ← ファンクション(マイナ)・コード DS：DX ← バッファへのポインタ	CF＝0：正常終了 CF＝1：AX ← エラー・コード
440DH	Generic IOCTL (for block devices)	一般 IOCTL(ブロック・デバイス用)	AX＝440DH BL ← デバイス番号 (00H＝デフォルト，01H＝A，…) CH＝08H：カテゴリ・コード CL ← ファンクション(マイナ)・コード DS：DX ← パラメータ・ブロック－1へのポインタ	CF＝0：正常終了 CF＝1：AX ← エラー・コード

〔表6-1〕ファンクション・リクエスト一覧 ⑥

番号	ファンクション名	機　能	コ　ー　ル	リ　タ　ー　ン
440EH	Get Logical Drive Map	論理ドライブ・マップの取得	AX=440EH BX←ドライブ番号 (00H=デフォルト，01H=A，…)	CF=0：正常終了 (AL=0：物理ドライブ) CF=1：AX←エラー・コード
440FH	Set Logical Drive Map	論理ドライブ・マップの設定	AX=440FH BX←ドライブ番号 (00H=デフォルト，01H=A，…)	CF=0：正常終了 (AL=0：物理ドライブ) CF=1：AX←エラー・コード
45H	Duplicate File Handle	ファイル・ハンドルの二重化	AH=45H BX←ファイル・ハンドル	CF=0：AX←新規のファイル・ハンドル CF=1：AX←エラー・コード
46H	Force Duplicate File Handle	ハンドルの強制二重化	AH=46H BX←既存のファイル・ハンドル CX←新規のファイル・ハンドル	CF=0：正常終了 CF=1：AX←エラー・コード
47H	Get Current Directory	カレント・ディレクトリの取得	AH=47H DS：SI←バッファ(64バイト)へのポインタ DL←ドライブ番号 (00H=カレント，01H=A，…)	CF=0：正常終了 CF=1：AX←エラー・コード
48H	Allocate Memory	メモリの割り当て	AH=48H BX←要求するメモリの大きさ (パラグラフ)	CF=0：AX←割り当てられたメモリのセグメント・アドレス CF=1：AX←エラー・コード BX←割り当て可能なメモリのパラグラフ・サイズ
49H	Free Allocated Memory	割り当てられたメモリの解放	AH=49H ES←解放すべきメモリ領域のセグメント・アドレス	CF=0：正常終了 CF=1：AX←エラー・コード
4AH	Set Block	割り当てられたメモリ・ブロックの変更	AH=4AH ES←メモリ領域のセグメント・アドレス BX←変更したいメモリの大きさ(パラグラフ)	CF=0：正常終了 CF=1：AX←エラー・コード BX←使用可能なメモリのパラグラフ・サイズ
4B00H	Load and Execute Program	プログラムのロードと実行	AX=4B00H DS：DX←パス名へのポインタ ES：BX←パラメータ・ブロックへのポインタ	CF=0：正常終了 CF=1：AX←エラー・コード
4B03H	Load Overlay	プログラム・セグメント(オーバレイ)のロード	AX=4B03H DS：DX←パス名へのポインタ ES：BX←パラメータ・ブロックへのポインタ	CF=0：正常終了 CF=1：AX←エラー・コード
4CH	End Process	プロセスの終了	AH=4CH AL←リターン・コード	なし
4DH	Get Return Code Child Process	子プロセスの終了コードの読み出し	AH=4DH	AH=00H：正常終了 AH=01H：ctrl-Cによる終了 AH=02H：致命的エラーによる終了 AH=03H：常駐終了 AL←リターン・コード
4EH	Find First File	最初に一致するファイルの検索	AH=4EH DS：DX←パス名へのポインタ CX←ファイルの属性	CF=0：正常終了 CF=1：AX←エラー・コード
4FH	Find Next File	次に一致するファイルの検索	AH=4FH	CF=0：正常終了 CF=1：AX←エラー・コード
54H	Get Verify State	ベリファイ・フラグの読み出し	AH=54H	AL=00H：フラグOFF AL=01H：フラグON
56H	Change Directory Entry	ディレクトリ・エントリの変更	AH=56H DS：DX←既存のファイルのパス名へのポインタ ES：DI←新規のパス名へのポインタ	CF=0：正常終了 CF=1：AX←エラー・コード
5700H	Get Date/Time of File	ファイルの日付/時刻の読み出し	AX=5700H BX←ファイル・ハンドル	CF=0：CX←時刻 DX←日付 CF=1：AX←エラー・コード

〔表6-1〕ファンクション・リクエスト一覧 ⑦

番号	ファンクション名	機　能	コ ー ル	リ タ ー ン
5701H	Set Date/Time of File	ファイルの日付/時刻の設定	AX=5701H BX←ファイル・ハンドル CX←設定すべき時刻 DX←設定すべき日付	CF=0：正常終了 CF=1：AX←エラー・コード
5800H	Get Allocation Strategy	メモリ割り当て方法の読み出し	AX=5800H	CF=0：AX←ストラテジ { 00H：下位 / 01H：最小 / 02H：上位 } CF=1：AX←エラー・コード
5801H	Set Allocation Strategy	メモリ割り当て方法の設定	AX=5801H BX←ストラテジ { 00H：下位 / 01H：最小 / 02H：上位 }	CF=0：正常終了 CF=1：AX←エラー・コード
59H	Get Extended Error	拡張されたエラー・コードの取得	AH=59H BX=0000H	AX←拡張されたエラー・コード BH←エラー・クラス BL←可能な対処 CH←エラーの発生箇所 CL, DX, SI, DI, BP, DS, ESの各レジスタは破壊される
5AH	Create Temporary File	一時ファイルの作成	AH=5AH CX←ファイルの属性 DS：DX←パス名へのポインタ(パス名の後に13バイト必要)	CF=0：AX←ファイル・ハンドル CF=1：AX←エラー・コード
5BH	Create New File	新しいファイルの作成	AH=5BH CX←ファイルの属性 DS：DX←パス名へのポインタ	CF=0：AX←ファイル・ハンドル CF=1：AX←エラー・コード
5C00H	Lock	ファイル・アクセスのロック	AX=5C00H BX←ファイル・ハンドル CX：DX←ロックする領域のオフセット SI：DI←ロックする領域の長さ	CF=0：正常終了 CF=1：AX←エラー・コード
5C01H	Unlock	ファイル・アクセスのロック解除	AX=5C01H BX←ファイル・ハンドル CX：DX←ロックを解除する領域のオフセット SI：DI←ロックを解除する領域の長さ	CF=0：正常終了 CF=1：AX←エラー・コード
5E00H	Get Machine Name	マシン名の取得	AX=5E00H DS：DX←バッファ(16バイト)へのポインタ	CF=0：CX←ローカル・コンピュータの番号 CF=1：AX←エラー・コード
5E02H	Printer Setup	プリンタの設定	AX=5E02H BX←割り当てリストのプリンタのインデックス CX←文字列の長さ DS：SI←文字列の先頭アドレス	CF=0：正常終了 CF=1：AX←エラー・コード
5F02H	Get Assign List Entry	割り当てリストのエントリ取得	AX=5F02H BX←割り当てリストのインデックス DS：SI←ローカル名バッファへのポインタ ES：DI←リモート名バッファへのポインタ	CF=0：BL=03H…プリンタ BL=04H…ドライブ CX←ユーザ変数域 CF=1：AX←エラー・コード
5F03H	Make Assign List Entry	割り当てリストのエントリ作成	AX=5F03H BL=03H：プリンタ 04H：ドライブ CX←ユーザ変数域 DS：SI←ソース・デバイス名のポインタ ES：DI←デスティネーション・デバイス名へのポインタ	CF=0：正常終了 CF=1：AX←エラー・コード
5F04H	Cancel Assign List Entry	割り当てリストのエントリ取り消し	AX=5F04H DS：SI←ソース・デバイス名へのポインタ	CF=0：正常終了 CF=1：AX←エラー・コード
62H	Get PSP	PSPの取得	AH=62H	BX←PSPのセグメント・アドレス

これに対して，ファンクション 2E〜62H は，ver. 2.11 以降になってサポートされた UNIX 指向のファンクション・リクエストです．なかでも，ファンクション 55H〜62H は，ver.3.10 以降の機能拡張にともなって追加されたファンクションです．これらのファンクションでは，ファイルとデバイスは同格化してファイル・ハンドルで扱われます．

CP/M コンパチブルな(ver.1.25)ファンクションでは，エラーの発生やその状態を AL レジスタに返します．これに対して ver.2.11 以降になってサポートされたファンクションでは，エラーの状態を CF(キャリ・フラグ)と AX レジスタに返します．

もし，エラーが発生すると CF に "1" を返し，そのエラー情報は**表6-2** のエラー・コードとして AX レジスタに返します．エラーがない場合は CF に "0" を返します．ここで，エラー・コードの 01H〜12H は MS-DOS ver.2.11 と互換性があり，13H〜58H は ver.3.10 において拡張されたエラー・コードです．

CP/M コンパチブルなファンクション・リクエストは，UNIX 指向のファンクション・リクエスト(ver. 2.11 以降)で代用できるため，特に必要のない限り後者のファンクション・リクエストを使用したほうが将来性の点からも賢明な方法といえます．

〔**表6-2**〕ver.2.11 以降のファンクション・リクエストにおけるエラー・コード

エラー・コード (AX レジスタ)	エラー内容	エラー・コード (AX レジスタ)	エラー内容
01H	無効なファンクション・コード	33H	リモート・コンピュータが LISTEN 状態にない
02H	ファイル名が見つからない	34H	ネットワーク名の二重定義
03H	パス名が無効	35H	ネットワーク名が見つからない
04H	オープンされているファイルが多すぎる	36H	ネットワークの準備ができていない
05H	アクセスが否定された	37H	これ以上のネットワーク・デバイスが存在しない
06H	無効なファイル・ハンドル	38H	ネットワーク BIOS の制限を起えた
07H	メモリ管理情報が破壊されている	39H	ネットワーク・アダプタのハード・エラー
08H	メモリ不足	3AH	ネットワークからの不正な応答
09H	メモリ・ブロック・アドレスが無効	3BH	予期できないネットワーク・エラー
0AH	不正な環境	3CH	ネットワーク・アダプタが適合しない
0BH	不正なフォーマット	3DH	プリント待ち行列が一杯である
0CH	無効なアクセス・コード	3EH	プリント待ち行列に空きがある
0DH	無効なデータ	3FH	プリント・ファイルのためのディスク領域が足りない
0EH	(予約)	40H	ネットワーク・デバイス名は削除されている
0FH	無効なドライブ名	41H	アクセスが拒絶された
10H	カレント・ディレクトリを削除しようとした	42H	ネットワーク・デバイスのタイプが不当
11H	無効なデバイス	43H	ネットワーク名が見つからない
12H	これ以上ファイルが存在しない	44H	ネットワーク名の制限を越えた
13H	ディスクがライト・プロテクト状態	45H	ネットワーク BIOS セッション数の制限を越えた
14H	不正なディスク・ユニット番号	46H	一時休止
15H	ドライブの準備ができていない	47H	ネットワークの要求が受け付けられない
16H	無効なディスク・コマンド	48H	プリンタまたはディスクのリダイレクション休止
17H	CRC エラー	49H〜4FH	(予約)
18H	コマンド・パケットの長さが違う	50H	同じ名前のファイルがすでに存在する
19H	シーク・エラー	51H	(予約)
1AH	MS-DOS フォーマットのディスクでない	52H	作成不能
1BH	セクタが見つからない	53H	INT 24H の失敗
1CH	プリンタの用紙切れ	54H	アサイン・リストの構造不良
1DH	書き込みエラー	55H	デバイス名は割り当て済み
1EH	読み出しエラー	56H	無効なパスワード
1FH	通常のエラー	57H	無効なパラメータ
20H	共有違反	58H	ネットワークへの書き込み失敗
21H	ロック違反		
22H	不正なディスク交換		
23H	FCB 使用不可能		
24H〜31H	(予約)		
32H	ネットワーク・リクエストが準備されていない		

6-1 ファンクション・リクエストの種類と呼び出し方法

ファンクション・リクエストの呼び出し方法としては，次のように三つの方法が用意されています．

INT 21H による方法

AH レジスタにファンクション番号(機能番号)をセットして

```
int 21h
```

を実行します．このとき，ほかのレジスタにはそれぞれのファンクションで指定される値をセットしなければなりません．

このINT 21Hによるファンクション・リクエスト(ファンクション59Hを除く)では，値が返されるレジスタ以外のレジスタはすべて保存されます(エラーによる場合も同様)．したがって，ユーザ・プログラム内ではこれらのレジスタの保存を考える必要はありません．

また，この方法が最もポピュラーな呼び出し方法となっています．

PSP を使用する方法

上と同様に各レジスタをセットし，

```
call   FAR PTR psp + 50h
```

を実行します．第3章で述べたように，PSPのオフセット50Hには，

```
int   21h
retf
```

の命令コードが入っているので，これにより同様のファンクション・リクエストが実行されます．

この方法を使用していると，将来，システム・コールの方法が変更(INT 21Hではなくなる)になった場合でも，プログラムの書き換えなしで対応できることになります．

CP/M 流の方法

ファンクション番号00H～24Hをコールしたい場合は，CP/M流の，

```
call   05h
```

によりファンクション・コールを行うことができます(COMモデルの場合)．この方法による場合は，CLレジスタにファンクション番号，DXレジスタに指定された値をセットします．

6-2 コンソール入出力関連のファンクション

コンソール入出力に関するファンクションとしては，**表6-3**のように10種類のファンクションが用意されています．これらは，エコーバック(キーボードからの入力文字を画面に表示すること)の有無やctrl-C入力の受け付けの有無などにより，それぞれ機能が異なっています．

また，これらの入出力は標準入出力を介して行われるので，I/Oリダイレクト機能によりコマンド・レベルでの入出力装置の変更やファイルへの割り当てが可能になっています．

〔表6-3〕コンソール入出力に関するファンクション

ファンクション番号	分類	機能	CP/M	MS-DOS ver.1.25	MS-DOS ver.2.11	MS-DOS ver.3.10	MS-DOS ver.3.30
01H	1文字入力	エコーバックあり，ctrl-C チェックあり	○	○	○	○	○
06H		エコーバックなし，ctrl-C チェックなし(出力も可)	○	○	○	○	○
07H		エコーバックなし，ctrl-C チェックなし	×	○	○	○	○
08H		エコーバックなし，ctrl-C チェックあり	×	○	○	○	○
02H	1文字出力	ctrl-C チェックあり	○	○	○	○	○
06H		ctrl-C チェックなし(入力も可)	○	○	○	○	○
0AH	文字列入力	テンプレート，漢字変換，ctrl-C チェックあり	○	○	○	○	○
0CH		バッファ・クリア，ALの内容によるファンクション・コールあり	×	○	○	○	○
09H	文字列出力	デリミタは "$"	○	○	○	○	○
0BH	ステータス	バッファのチェック，ctrl-C チェックあり	○	○	○	○	○

Read Keyboard and Echo	01H
機能	キーボード入力とエコー
コール	AH＝01H
リターン	AL←入力された文字コード

標準入力から1文字入力されるまで待ち，入力された文字を標準出力にエコー・バックします．また，その文字コードはALレジスタに返されます．入力文字がctrl-Cの場合は割り込みタイプ23H(127ページ)を実行します．

Display Character	02H
機能	文字の出力
コール	AH＝02H DL←出力すべき文字コード
リターン	なし

DLレジスタ内の文字を標準出力に出力します．ctrl-Cが入力された場合は，割り込みタイプ23Hが実行されます．

Direct Console Input	06H
機能	コンソールからの直接入力
コール	AH＝06H DL＝FFH
リターン	ZF＝1 AL＝00H(入力なし) ZF＝0 AL←入力した文字コード

標準入力から文字コードを入力します．このファンクションではctrl-C入力のチェックは行われません．

Direct Console Output	06H
機能	コンソールへの直接出力
コール	AH＝06H DL←FFH以外の文字コード
リターン	なし

標準出力に文字を出力します．このファンクションではctrl-C入力のチェックは行われません．

ファンクション06Hは，DLレジスタに与えるデータによって，入力と出力の両方に使用できます．

Direct Console Input	07H
機能	直接コンソール入力
コール	AH＝07H
リターン	AL←標準入力から入力された文字コード

標準入力から文字が入力されるまで待ち，この入力された文字コードをALレジスタに返します．文字のエコーバックやctrl-C入力のチェックは行いません．

Read Keyboard	08H
機能	キーボード入力
コール	AH＝08H
リターン	AL←標準入力から入力された文字コード

標準入力から1文字入力されるまで待ち，その文字コードをALレジスタに返します．このファンクションでは，ctrl-C入力による割り込みは実行しますが，文字のエコーバックは行いません．

Display String	09H
機能	文字列の出力
コール	AH＝09H DS：DX←出力する文字列の先頭アドレス
リターン	なし

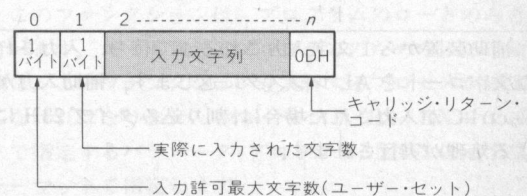
文字列の格納されている先頭アドレスをDS：DXにセットしてファンクション・リクエストを行います．文字列の最後は“$”で表しますが，この$は出力されません．また，このファンクションの実行中にはctrl-C入力のチェックも行われます．

Buffered Keyboard Input	0AH
機能	バッファード・キーボード入力
コール	AH＝0AH DS：DX←入力バッファの先頭アドレス
リターン	なし

標準入力からの入力を，リターン・キーが押されるまで入力バッファの先頭アドレス＋2番地から順次格納していきます．

なお，バッファのフォーマットは図6-1のようになっていて，入力文字が指定された文字数より多く入力された場合，その越えた分は無視されてASCIIコードのBEL(07H)が標準出力に出力されます．

バッファの先頭アドレス＋1番地には，CRコード(0DH)を含まない入力文字数がセットされて返されます．このシステム・コールではctrl-Cのチェックも行われ，またテンプレート機能も利用することが可能です．

〔図6-1〕入力バッファのフォーマット

Check Keyboard Status		0BH
機能	キーボード・ステータスのチェック	
コール	AH＝0BH	
リターン	AL＝FFH タイプ・アヘッド・バッファに文字が入っている	
	AL＝00H タイプ・アヘッド・バッファは空である	

タイプ・アヘッド・バッファ内に文字が入っているかどうかを検査します．ctrl-C がバッファ内に入っている場合には割り込みタイプ 23H が実行されます．

Flush Buffer,Read Keyboard		0CH
機能	バッファを空にしてキーボード入力	
コール	AH＝0CH	
	AL＝01H, 06H, 07H, 08H, 0AH(対応するファンクション・リクエストが行われる)	
	AL←それ以外の値(タイプ・アヘッド・バッファを空にする)	
リターン	AL＝00H タイプ・アヘッド・バッファは空になっている	

キーボードのタイプ・アヘッド・バッファを空にします．このとき，AL レジスタに設定されている数値によって対応するファンクションがリクエストされます．

6-3 外部入出力に関するファンクション

補助入出力装置(RS-232C など)やプリンタとの文字の転送を行うファンクションで，**表6-4** に示すように 3 種類が用意されています．

これらの装置へのアクセスは，標準ファイル・ハンドルによっても行うことができます．また，これらのファンクションではデバイスに対する入出力を行うため，エラーの発生も考えられますがエラーは返されません．

Auxiliary Input		03H
機能	補助入力	
コール	AH＝03H	
リターン	AL←補助装置から入力される文字コード	

補助装置から 1 文字入力されるまで待ち，入力された文字コードを AL レジスタに返します．補助入力から ctrl-C が入力された場合は，割り込みタイプ 23H による処理が実行されます．

Auxiliary Output		04H
機能	補助出力	
コール	AH＝04H	
	DL←補助装置に出力すべき文字コード	
リターン	なし	

DL レジスタ内の文字を補助出力装置に出力します．もし，標準入力装置から ctrl-C が入力された場合は割り込みタイプ 23H が実行されます．

Print Character		05H
機能	プリンタへの文字の出力	
コール	AH＝05H	
	DL←プリンタに出力すべき文字コード	
リターン	なし	

DL レジスタ内の文字をプリンタに出力します．もし，標準入力から ctrl-C が入力されると，内部割り込みタイプ 23H が実行されます．

6-4 プロセス管理に関するファンクション

プログラムの実行や終了に関するファンクションは，**表6-5** に示すように，サブファンクションも含めて 8 種類が用意されています．これらのうち，ファンクション 00H 以外は，階層プロセスをサポートするために ver.2.11 になってから追加されたものです．

Terminate Program		00H
機能	プログラムの終了	
コール	AH＝00H	
	CS←PSP のセグメント・アドレス	
リターン	なし	

このファンクション・リクエストは，内部的には割り込みタイプ 20H をコールしますので，この処理の内容や注意事項については，前章の割り込みタイプ 20H の項(127 ページ)を参照してください．

Create New PSP		26H
機能	新しい PSP の作成	
コール	AH＝26H	
	DX←新しい PSP のセグメント・アドレス	
リターン	なし	

このファンクションでは，DX レジスタで指定されたセグメント・アドレスに新しい PSP を作成します．なお，このファンクションは，ver.2.0 以前の MS-DOS との互換性を保つために用意されているもので，新し

〔表6-4〕外部入出力装置に関するファンクション

ファンクション番号	機能	CP/M	MS-DOS ver.1.25	MS-DOS ver.2.11	MS-DOS ver.3.10	MS-DOS ver.3.30
03H	補助装置からの入力	○	○	○	○	○
04H	補助装置への出力	○	○	○	○	○
05H	プリンタへの出力	○	○	○	○	○

〔表6-5〕プロセス管理に関するファンクション

ファンクション番号	分類	機能	CP/M	MS-DOS ver.1.25	MS-DOS ver.2.11	MS-DOS ver.3.10	MS-DOS ver.3.30
00H	プロセス終了	プロセス終了(COM モデル)	○	○	○	○	○
31H		プロセス常駐終了	×	×	○	○	○
4CH		プロセス終了(全モデル)	×	×	○	○	○
4DH		プロセス終了コード読み出し	×	×	○	○	○
4B00H	プロセス・ロード	プロセスのロードと実行	×	×	○	○	○
4B03H		プロセスのロード(オーバレイ)	×	×	○	○	○
26H	PSP	新しい PSP の作成	×	×	×	○	○
62H		PSP セグメント・アドレスの取得	×	×	×	○	○

いプログラムではファンクション 4B00H(子プロセスの起動)を使用すべきです．

Keep Process	31H
機能	プログラムの常駐終了
コール	AH＝31H AL ←リターン・コード DX ←常駐させるプログラムのパラグラフ・サイズ
リターン	なし

現在のプロセスをメモリに常駐させて終了します．このとき，DX レジスタで指定された大きさのパラグラフ・サイズがプログラム領域として確保されます．

AL レジスタでセットしたリターン・コードは，ファンクション 4DH により親プロセスで取得することが可能です．

Load and Execute a Program	4B00H
機能	プログラムのロード/実行
コール	AX＝4B00H DS：DX ←パス名の先頭アドレス ES：BX ←パラメータ・ブロックの先頭アドレス
リターン	CF＝1 の場合 AX ←エラー・コード(**表6-2**) CF＝0 の場合 正常終了

子プロセスのロードと実行を行います．ファイル名は ASCIZ 文字列でセットします．パラメータ・ブロックは**図6-2** に示したフォーマットで指定します．

〔図6-2〕ファンクション 4B00H のパラメータ・ブロックのフォーマット

ES：BX	オフセット	サイズ	内容
→	+00H	1ワード	環境変数セグメント・アドレス
	+02H	2ワード	パラメータ文字列のアドレス
	+06H	2ワード	第1 FCB 文字列のアドレス
	+0AH	2ワード	第2 FCB 文字列のアドレス

Load a Program	4B03H
機能	プログラムのロード
コール	AX＝4B03H DS：DX ←パス名の先頭アドレス ES：BX ←パラメータ・ブロックの先頭アドレス
リターン	CF＝1 の場合 AX ←エラー・コード(**表6-2**) CF＝0 の場合 正常終了

このファンクションは，プログラムのロードのみを行います．ロードされるプログラムのためのメモリ領域は，このファンクションを実行するプロセスが確保しなければなりません(オーバレイ)．ES：BX レジスタで指定するパラメータ・ブロックは，**図6-3** に示すフォーマットで指定します．

〔図6-3〕ファンクション 4B03H のパラメータ・ブロックのフォーマット

ES：BX		
+00H	1ワード	プログラムがロードされるセグメント・アドレス
+02H	1ワード	リロケーション要素（通常はプログラムがロードされるセグメント・アドレス）

Terminate a Process	4CH
機能	プロセスの終了
コール	AH＝4CH AL←リターン・コード
リターン	なし

現在のプロセスを終了して親プロセスに戻ります．このとき，AL レジスタに設定されているリターン・コードは，ファンクション 4DH を用いて親プロセスで参照することができます．

Get Return Code of a Child Process	4DH
機能	子プロセスの終了コードの読み出し
コール	AH＝4DH
リターン	AH＝00H：終了(ファンクション 00H, 4CH, INT 20H) AH＝01H：ctrl-C による終了 (INT 23H) AH＝02H：致命的エラーによる終了 (INT 24H) AH＝03H：常駐したまま終了(ファンクション 31H, INT 27H) AL←リターン・コード

このファンクションによって，子プロセスがどのような状態で終了したのか，そのリターン・コードから知ることができます．

Get PSP	62H
機能	PSP のセグメント・アドレスの取得
コール	AH＝62H
リターン	BX＝カレント・プロセスの PSP のセグメント・アドレス

このファンクションは，現在実行中のプロセス(自分自身)の PSP のセグメント・アドレスを返します．

6-5 メモリ管理に関するファンクション

メモリの管理に関するファンクションは，**表6-6** のようにサブファンクションも含めて5種類が用意されています．これらは，すべて ver.2.11 になってから追加されたものです．

Allocate Memory	48H
機能	メモリ・ブロックの割り当て
コール	AH＝48H BX←割り当てられるメモリ領域のパラグラフ・サイズ
リターン	CF＝1 の場合 AX←エラー・コード(**表6-2**) BX←割り当て可能な最大メモリ領域のパラグラフ・サイズ CF＝0 の場合 AX←割り当てられたメモリ領域の先頭のセグメント・アドレス

BX レジスタによって指定されたメモリを割り当てます．MS-DOS では，一般にユーザ・プログラムに対してメモリの終わりまでが割り当てられます．このため，このファンクションを実行するまえに，次に示すファンクション 49H や 4AH によって，メモリの空き領域を確保しなければなりません．

〔表6-6〕メモリ管理に関するファンクション

ファンクション番号	機　　能	CP/M	MS-DOS ver.1.25	MS-DOS ver.2.11	MS-DOS ver.3.10	MS-DOS ver.3.30
48H	メモリ・ブロックの割り当て	×	×	○	○	○
49H	メモリ・ブロックの解放	×	×	○	○	○
4AH	メモリ・ブロックのサイズ変更	×	×	○	○	○
5800H	メモリ・アロケーション・ストラテジの設定	×	×	×	○	○
5801H	メモリ・アロケーション・ストラテジの取得	×	×	×	○	○

Free Allocated Memory	49H
機能	メモリ・ブロックの解放
コール	AH＝49H ES←解放すべきメモリ領域のセグメント・アドレス
リターン	CF＝1の場合 AX←エラー・コード(表6-2) CF＝0の場合 正常終了

指定されたメモリ・ブロックをMS-DOSの管理下に戻します.

Modify Allocated Memory Blocks	4AH
機能	メモリ・ブロックのサイズの変更
コール	AH＝4AH ES←メモリ領域のセグメント・アドレス BX←割り当てを要求するメモリ領域のセグメント・アドレス
リターン	CF＝1の場合 AX←エラー・コード(表6-2) BX←使用可能な最大のメモリ・サイズ(パラグラフ) CF＝0の場合 正常終了

割り当てられているメモリ・ブロックの大きさを拡大または縮小します.

Get Allocation Strategy	5800H
機能	メモリ・アロケーション・ストラテジの取得
コール	AX＝5800H
リターン	CF＝1の場合 AX←エラー・コード(表6-2) CF＝0の場合 AX←ストラテジ

このファンクションは，メモリを割り当てる際のストラテジを取得します．ストラテジの値と意味は次のとおりです．なおシステム起動時には下位(00H)となっています.

00H：下位

割り当てられるメモリが可能な限り下位に位置するようにする.

01H：最小

要求を満たすメモリ領域のうち最小のメモリ領域を割り当てる.

02H：上位

割り当てられるメモリが可能な限り上位に位置するようにする.

Set Allocation Strategy	5801H
機能	メモリ・アロケーション・ストラテジの設定
コール	AX＝5801H BX←ストラテジ
リターン	CF＝1の場合 AX←エラー・コード(表6-2) CF＝0の場合 正常終了

このファンクションは，メモリを割り当てる際のストラテジを設定します．ストラテジの値と意味はファンクション5800Hと同じです.

6-6 タイマ設定に関するファンクション

リアルタイム・クロックの設定に関するファンクションは表6-7のように4種類に分類されます．これらの操作の大部分は，command.comの内部コマンドによって実現されています.

Get Date	2AH
機能	日付の読み出し
コール	AH＝2AH
リターン	CX←年(1980～2099) DH←月(1～12) DL←日(1～31) AL←曜日(0＝日，1＝月，…，6＝土)

上記のレジスタに，現在の日付が2進数表現で返さ

〔表6-7〕タイマ設定に関するファンクション

ファンクション番号	分類	機能	CP/M	MS-DOS ver.1.25	MS-DOS ver.2.11	MS-DOS ver.3.10	MS-DOS ver.3.30
2AH	日付	日付の読み出し	×	○	○	○	○
2BH		日付の設定	×	○	○	○	○
2CH	時刻	時刻の読み出し	×	○	○	○	○
2DH		時刻の設定	×	○	○	○	○

れます．

Set Date		2BH
機能	日付の設定	
コール	AH＝2BH	
	CX ←年(1980～2099)	
	DH ←月(1～12)	
	DL ←日(1～31)	
リターン	AL＝00H　日付はセットされた	
	AL＝FFH　日付は無効	

CX および DX レジスタは，2 進数で表現された有効な日付がセットされていなければなりません．曜日は自動的に計算されます．

Get Time		2CH
機能	時刻の読み出し	
コール	AH＝2CH	
リターン	CH ←時(0～23)	
	CL ←分(0～59)	
	DH ←秒(0～59)	
	DL ←1/100 秒(0～99)	

現在の時刻を 2 進数表現で CX，DX レジスタに返します．PC-9801 の場合は 1/100 秒は管理されていないので，DL レジスタには常に 00H が返されます．

Set Time		2DH
機能	時刻の設定	
コール	AH＝2DH	
	CH ←時(0～23)	
	CL ←分(0～59)	
	DH ←秒(0～59)	
	DL ← 1/100 秒(0～99)	
リターン	AH＝00H　時刻がセットされた	
	AH＝FFH　時刻が無効	

CX および DX レジスタには 2 進数表現された有効な時刻が入っていなければなりません．PC-9801 の場合は 1/100 秒は管理されていませんが，DL レジスタには 00H をセットしなければなりません．

6-7 システム設定に関するファンクション

各種のフラグや割り込みベクタの操作など，システムの設定に関するファンクションは，**表6-8** のようにサブファンクションを含めて 9 種類が分類されます．これらの操作の大部分は，command.com の内部コマンドにより実現されています．

Set Vector		25H
機能	割り込みベクタの設定	
コール	AH＝25H	
	AL←割り込みタイプ番号	
	DS：DX ←割り込みルーチン・アドレス	
リターン	なし	

このファンクションは，特定の割り込みタイプのベクタを設定するために使用されます．これにより，致命的(ハード)エラー処理や，ctrl-C 入力によるプログラム中断処理などの処理ルーチンを，ユーザ・プログラムの中にもつことも可能です．

Set/Reset Verify Flag		2EH
機能	ベリファイ・フラグの設定	
コール	AH＝2EH	
	AL＝00H　ベリファイを行わない	
	AL＝01H　ベリファイを行う	
リターン	なし	

verify コマンドと同様に，ベリファイ・フラグの

〔表6-8〕 システム設定に関するファンクション

ファンクション番号	分類	機能	CP/M	MS-DOS ver.1.25	MS-DOS ver.2.11	MS-DOS ver.3.10	MS-DOS ver.3.30
25H	割り込みベクタ	割り込みベクタの設定	×	○	○	○	○
35H		割り込みベクタの読み出し	×	×	○	○	○
2EH	ベリファイ・フラグ	ベリファイ・フラグの設定	×	×	○	○	○
54H		ベリファイ・フラグの読み出し	×	×	○	○	○
30H	バージョン	DOS バージョン番号の読み出し	×	×	○	○	○
3300H	BREAK チェック・フラグ	BREAK チェック・フラグの読み出し	×	×	○	○	○
3301H		BREAK チェック・フラグの設定	×	×	○	○	○
38H	カントリ	国別情報の設定/読み出し	×	×	○	○	○
59H	エラー・コード	拡張エラー・コードの読み出し	×	×	×	○	○

ON/OFFを指定します．ベリファイONではディスクに書き込みを行う際に，その内容が正しく書き込まれたかどうかの検査を行います．

ただし，このベリファイを行うとディスクへの書き込み終了までに時間がかかります．したがって，重要なデータでない場合はベリファイ・フラグをOFFにしておいたほうがよいでしょう．

Get DOS Version Number	30H
機能	DOSのバージョン番号の読み出し
コール	AH＝30H
リターン	AL←バージョン番号の整数部
	AH←バージョン番号の少数部
	BH←OEMのシリアル番号
	BL：CX←24ビットのユーザ番号

MS-DOSのバージョン番号を返します．たとえば，MS-DOS ver.2.11の場合はAL＝02H，AH＝0BH(すなわち11)が返されます．

ctrl-C Check	3300H
機能	BREAKチェック・フラグの読み出し
コール	AX＝3300H
リターン	DL＝00H　フラグ　OFF
	DL＝01H　フラグ　ON

breakコマンドと同様に，ディスク・アクセスなども含むすべてのシステム・コールにおいて，ctrl-C入力のチェックを行うかどうかのフラグの読み出しを行います．

結果はDLレジスタに返され，フラグがOFFのときにはDL＝00Hであり，フラグがONのときはDL＝01Hとなります．

ctrl-C Check	3301H
機能	BREAKチェック・フラグの設定
コール	AX＝3301H
	DL＝00H　フラグ　OFF
	DL＝01H　フラグ　ON
リターン	AL＝FFH　フラグの設定が無効

breakコマンドと同様に，ディスク・アクセスなども含むすべてのシステム・コールにおいて，ctrl-C入力のチェックを行うかどうかのフラグの設定を行います．

Get Interrupt Vector	35H
機能	割り込みベクタの読み出し
コール	AH＝35H
	AL←割り込みタイプ番号
リターン	ES：BX←割り込みルーチン・アドレス

指定した割り込みタイプのベクタを返します．

Get Country Data	38H
機能	国別情報の読み出し
コール	AH＝38H
	AL＝00H　現在の国
	AL＝01H　USAの規格
	AL＝51H　日本の規格
	DS：DX←バッファ(32バイト)の先頭アドレス
リターン	CF＝1の場合
	AX←エラー・コード(表6-2)
	CF＝0の場合
	バッファに国別情報がセットされる

エラーがない場合には，バッファに対して図6-4に示すようなフォーマットの情報が返されます．

〔図6-4〕国別情報のフォーマット(38H)

Set Country Data	38H
機能	国別情報の設定
コール	AH＝38H DX＝FFFFH AL←カントリ・コード(FFH 以外) BH←FFH 以上のカントリ・コード (AL＝FFH のとき)
リターン	CF＝1 の場合 AX←エラー・コード(**表6-2**) CF＝0 の場合 正常終了

このファンクションでは，カントリ・コードを用いて国別情報の設定を行います．カントリ・コードについてはファンクション 38H の読み出し(Get Country Data)と同じです．

Return Current Setting of Verify Flag	54H
機能	ベリファイ・フラグの読み出し
コール	AH＝54H
リターン	AL←00H　ベリファイ・フラグは OFF AL←01H　ベリファイ・フラグは ON

現在のベリファイ・フラグの設定状況が AL レジスタに報告されます．

Get Extended Error	59H
機能	拡張エラー・コードの取得
コール	AH＝59H BX←0000H
リターン	AX←拡張エラー・コード BH←エラー・クラス BL←可能な処置 CH←ローカス CL，DX，SI，DI，BP，DS，ES レジスタは破壊される

ver.2.11 までのファンクション・リクエストで返されるエラー・コードや，INT 24H ハンドラにおいて DI レジスタに返されるエラー・コードは，ver.2.11 との互換性を保つために 00H～12H に集約されています．このファンクションを利用すると，ver.3.10 以降に対応した詳しい拡張エラー・コード(00H～58H；155 ページ，**表6-2**)を得ることができます．

BH レジスタに返されるエラー・クラスは**表6-9** のように定義されています．

BL レジスタに返される可能な処理は，**表6-10** のように定義されていて，プログラムが対応すべき処理を表すコードです．

CH レジスタに返されるローカスは，**表6-11** のように定義されていて，エラーの発生した箇所を表します．

〔表6-9〕エラー・クラス(BH レジスタ)

コード	内　容
01H	メモリ容量や I/O チャネルなどの資源不足
02H	エラーではないがファイルの一部がロックされているなどプロセスを終了すべき状況にある
03H	パーミッションに関するエラー
04H	システム・ソフトウェアの内部エラー
05H	ハードウェアに起因するエラー
06H	現在のプロセスが原因ではないシステム・ソフトウェアのエラー
07H	アプリケーション・プログラムのエラー
08H	ファイルが存在しない
09H	無効なフォーマット
0AH	ファイルが内部的にロックされている
0BH	ディスクの不良
0CH	その他の原因によるエラー

〔表6-10〕可能な対処(BL レジスタ)

コード	対　処
01H	ユーザに確認を求め，再試行する
02H	休止後に再試行する
03H	ユーザに再度入力を求める
04H	メモリ内容をクリアして終了する
05H	ただちにプログラムを終了する
06H	エラー・コードに対応する処理を行う
07H	ユーザ側でディスク交換などの対処を行う

〔表6-11〕ローカス(CH レジスタ)

コード	箇　所
01H	不明
02H	ブロック・デバイス
03H	ネット・ワーク
04H	キャラクタ・デバイス
05H	メモリ・エラー

6-8 ディスク管理に関するファンクション

これらのファンクションは，ディスク・ドライブの設定やバッファの設定などに関するファンクションで，**表6-12** のように 8 種類が分類されます．

Disk Reset	0DH
機能	ディスクのリセット
コール	AH＝0DH
リターン	なし

このファンクションにより，すべてのディスク・バ

〔表6-12〕ディスク管理に関するファンクション

ファンクション番号	分類	機能	CP/M	MS-DOS ver.1.25	MS-DOS ver.2.11	MS-DOS ver.3.10	MS-DOS ver.3.30
0DH	ドライブ	ディスクのリセット(バッファの解放)	○	○	○	○	○
0EH		ドライブの選択	○	○	○	○	○
19H		カレント・ドライブの読み出し	○	○	○	○	○
1BH	ディスク・データ	カレント・ドライブのデータ読み出し	×	×	×	○	○
1CH		指定ドライブのデータ読み出し	×	×	×	○	○
36H		ディスク残り容量の読み出し	×	×	○	○	○
1AH	ディスク転送アドレス	ディスク転送アドレスの設定	○	○	○	○	○
2FH		ディスク転送アドレスの読み出し	×	×	○	○	○

ッファの解放を行います.

CP/M では,ディスクを交換すると,そのディスク・ドライブは読み出し専用になるため,このファンクションにより書き込み可能にしていました.しかし,MS-DOS では常に読み出し/書き込みが可能なので,通常のユーザ・プログラムではこのファンクションは必要ありません.

Select Disk	0EH
機能	ドライブの選択
コール	AH＝0EH DL←ドライブ番号 (00H＝A, 01H＝B, …)
リターン	AL←論理ドライブの数

DL レジスタで指定されたドライブがカレント・ドライブとして選択され,接続されているドライブの数が AL レジスタに返されます.無効なドライブを指定した場合は何も行われません.

Get Current Disk	19H
機能	カレント・ドライブ番号の読み出し
コール	AH＝19H
リターン	AL←現在選択されているドライブ番号 (00H＝A, 01H＝B, …)

このファンクションは,プログラムの中でカレント・ドライブがどこにあるのかを知りたい場合に使用されます.

Set Disk Transfer Address	1AH
機能	ディスク転送アドレスの設定
コール	AH＝1AH DS：DX←ディスク転送アドレス
リターン	なし

ディスク転送アドレス(Disk Transfer Address：以下 DTA)の設定を行います.DTA とは,FCB を用いてディスクの読み出し/書き込みを行う際のバッファや,ファイル検索を行う際のバッファのアドレスをいいます.

このファンクションで指定するバッファは,セグメントの境界に跨ってはいけません.また,このファンクションで DTA の位置を指定しない場合は,PSP のオフセット 80H～FFH の 128 バイトがデフォルトの DTA として使用されます.

Get Default Drive Data	1BH
機能	カレント・ドライブのデータの取得
コール	AH＝1BH
リターン	AL←1クラスタ当たりのセクタ数 CX←1セクタ当たりのバイト数 DX←1ドライブ当たりのクラスタ数 DS：BX←FAT-ID の書かれているアドレス

このファンクションは,カレント・ドライブに挿入されているディスクに関する情報を得るためのファンクションです.

Get Drive Data	1CH
機能	指定ドライブのデータの取得
コール	AH＝1CH DL←ドライブ番号 (00H＝カレント, 01H＝A, …)
リターン	AL←FFH：ドライブ番号の指定が無効 FFH 以外：1クラスタ当たりのセクタ数 CX←1セクタ当たりのバイト数 DX←1ドライブ当たりのクラスタ数 DS：BX←FAT-ID の書かれているアドレス

このファンクションは,指定したドライブに挿入さ

れているディスクに関する情報を得るためのファンクションです．

Get Disk Transfer Address	2FH
機能	ディスク転送アドレスの読み出し
コール	AH＝2FH
リターン	ES：BX←ディスク転送アドレス

このファンクションは，DTA(ディスク転送用バッファ)のアドレスをES：BXレジスタに返します．デフォルトのDTAはPSP内のオフセット80Hからの128バイトです．

Get Disk Free Space	36H
機能	ディスク残り領域の読み出し
コール	AH＝36H DL←ドライブ番号 (00H＝カレント，01H＝A，…)
リターン	BX←使用可能なクラスタ数 DX←1ドライブ当たりのクラスタ数 CX←1セクタ当たりのバイト数 AX←1クラスタ当たりのセクタ数 AX＝FFFFH　ドライブ番号が無効

DLレジスタで指定されたドライブのディスク情報が返されます．使用可能なバイト数は次式から計算できます．

使用可能なバイト数
＝使用可能なクラスタ数×1クラスタ当たりのセクタ数×1セクタ当たりのバイト数

6-9 FCB関係のファンクション

FCBによるファイル・アクセスを行う際に使用されるファンクションは，**表6-13**に示すように16種類のファンクションが用意されています．

これらのファンクションのほとんどは，CP/M(またはMS-DOS ver.1.25)から受け継がれたものです．

Open a File	0FH
機能	ファイルのオープン
コール	AH＝0FH DS：DX←オープンされていないFCBの先頭アドレス
リターン	AL＝00H　ファイルが存在する AL＝FFH　ファイルが存在しない

指定されたファイルがオープンされます．このとき，FCBの中の次のフィールドがセットされます．

(1) ドライブ番号
(2) カレント・ブロック
(3) レコード・サイズ
(4) ファイル・サイズ
(5) 日付と時刻

また，このあとに読み出しや書き込みを行う場合は，次のフィールドをセットする必要があります．

(1) シーケンシャル・ファイルの場合：カレント・レコード
(2) ランダム・ファイルの場合：相対レコード

〔表6-13〕FCB関係のファンクション

ファンクション番号	分類	機能	CP/M	MS-DOS ver.1.25	MS-DOS ver.2.11	MS-DOS ver.3.10	MS-DOS ver.3.30
11H	ファイル名	最初のファイルの検索	○	○	○	○	○
12H		次のファイルの検索	○	○	○	○	○
13H		ファイルの削除	○	○	○	○	○
16H		ファイルの作成	○	○	○	○	○
17H		ファイル名の変更	○	○	○	○	○
29H		ファイル名の解析	×	○	○	○	○
0FH	ファイルのアクセス	ファイルのオープン	○	○	○	○	○
10H		ファイルのクローズ	○	○	○	○	○
14H		シーケンシャル読み出し	○	○	○	○	○
15H		シーケンシャル書き込み	○	○	○	○	○
21H		ランダム読み出し	○	○	○	○	○
22H		ランダム書き込み	○	○	○	○	○
27H		ランダム・ブロック読み出し	×	○	○	○	○
28H		ランダム・ブロック書き込み	×	○	○	○	○
23H	計算	ファイル・サイズの計算	○	○	○	○	○
24H		相対レコード・フィールドの設定	○	○	○	○	○

Close a File		10H
機能	ファイルのクローズ	
コール	AH=10H	
	DS:DX←オープンされているFCBの先頭アドレス	
リターン	AL=00H	ファイルが存在する
	AL=FFH	ファイルが存在しない

ファイル内容が変更された場合，ディレクトリ・エントリを更新するため，このファンクションを実行しなければなりません．

Search for First Entry		11H
機能	最初のファイルの検索	
コール	AH=11H	
	DS:DX←オープンされていないFCBの先頭アドレス	
リターン	AL=00H	ファイルが存在する
	AL=FFH	ファイルが存在しない

カレント・ディレクトリを検索して，FCBで指定されたファイル名(ワイルド・カードも含む)に一致するファイルがあれば，そのファイルの情報をFCBと同じ形式でDTAにセットします．

ワイルド・カードを用いた検索では，実際のファイル名がDTAに返されます．

Search for Next Entry		12H
機能	次のファイルの検索	
コール	AH=12H	
	DS:DX←オープンされていないFCBの先頭アドレス	
リターン	AL=00H	ファイルが存在する
	AL=FFH	ファイルが存在しない

ファンクション11Hにより検索されたファイルを続けて検索します．指定されたファイル名(ワイルド・カードを含む)に一致するファイルが存在すれば，ファンクション11Hと同様に，DTA内にその情報がセットされます．

Delete File		13H
機能	ファイルの削除	
コール	AH=13H	
	DS:DX←オープンされていないFCBの先頭アドレス	
リターン	AL=00H	ファイルが存在した
	AL=FFH	ファイルが存在しない

FCBで指定されたファイルが削除されます．この場合に，ファイル名にはワイルド・カードを使用することができ，該当するすべてのファイルを削除することができます．

Sequential Read		14H
機能	シーケンシャルな読み出し	
コール	AH=14H	
	DS:DX←オープンされているFCBの先頭アドレス	
リターン	AL=00H	読み出しは正常に行われた
	AL=01H	ファイルの終わり(EOF)を検出した．このレコードにデータは入っていない
	AL=02H	DTAが小さ過ぎる
	AL=03H	読み出しの途中でEOFを検出した．EOFまでのデータがDTAに読み込まれ，残りの部分は00Hで埋められる

ファイルからレコード・サイズに等しいバイト数がDTA(ディスク転送アドレス)に読み込まれ，FCB内のカレント・ブロックやカレント・レコードが次のブロックを指すようにインクリメントされます．

Sequential Write		15H
機能	シーケンシャルな書き込み	
コール	AH=15H	
	DS:DX←オープンされているFCBの先頭アドレス	
リターン	AL=00H	書き込みは正常に行われた
	AL=01H	ディスクに空き領域がない
	AL=02H	DTAが小さ過ぎる

DTAからレコード・サイズに等しいバイト数がファイルに書き込まれます．このあと，カレント・ブロックやカレント・レコードが次のブロックを指すようにインクリメントされます．

Create a File		16H
機能	ファイルの作成	
コール	AH=16H	
	DS:DX←オープンされていないFCBの先頭アドレス	
リターン	AL=00H	ファイルが作成された
	AL=FFH	空のディレクトリ・エントリが存在しない

指定されたファイルが作成されオープンされます．このとき，拡張FCBを使用して隠されたファイルを作成することも可能です．既存のファイルがあった場合は，そのファイル内容は失われ，大きさが0バイトのファイルになります．

Rename a File		17H
機能	ファイル名の変更	
コール	AH＝17H	
	DS：DX←変更すべきファイル名が入ったFCBの先頭アドレス	
リターン	AL＝00H	ファイルが存在する
	AL＝FFH	目的のファイル名が存在しないか，新しいファイル名がすでに存在する

rename コマンドと同様に新しいファイル名がすでに存在してはなりません．ファイル名にはワイルド・カード(？のみ)を使用することができます．

Random Read		21H
機能	ランダムな読み出し	
コール	AH＝21H	
	DS：DX←オープンされているFCBの先頭アドレス	
リターン	AL＝00H	読み出しは正常に行われた
	AL＝01H	ファイルの終わり(EOF)を検出した．このレコードにデータは入っていない
	AL＝02H	DTAが小さ過ぎる
	AL＝03H	読み出しの途中でEOFを検出した．EOFまでのデータがDTAに読み込まれ，残りの部分は00Hで埋められる

カレント・ブロックとカレント・レコードの値が，相対レコードで指定されたレコードと一致するようにセットされ，次にレコード・サイズで指定されたバイト数がDTAに読み込まれます．

このファンクションでは，カレント・ブロックやカレント・レコードは自動的にはインクリメントされません．

Random Write		22H
機能	ランダムな書き込み	
コール	AH＝22H	
	DS：DX←オープンされているFCBの先頭アドレス	
リターン	AL＝00H	書き込みは正常に行われた
	AL＝01H	ディスクに空き領域がない
	AL＝02H	DTAが小さ過ぎる

カレント・ブロックとカレント・レコードの値が，相対レコードで指定されたレコードと一致するようにセットされ，次にレコード・サイズで指定されたバイト数がDTAからファイルに書き込まれます．

このファンクションでは，カレント・ブロックやカレント・レコードは自動的にはインクリメントされません．

File Size		23H
機能	ファイル・サイズの計算	
コール	AH＝23H	
	DS：DX←オープンされていないFCBの先頭アドレス	
リターン	AL＝00H	ファイルが存在する
	AL＝FFH	ファイルが存在しない

ファイルの最終レコードを相対レコード・フィールドにセットします．

Set Relative Record		24H
機能	相対レコードの設定	
コール	AH＝24H	
	DS：DX←オープンされたFCBの先頭アドレス	
リターン	なし	

FCBで指定されたファイルが見つかると，相対レコード・フィールドが，カレント・ブロックとカレント・レコードが指しているファイル・アドレスと同じアドレスを指すように計算されてセットされます．

Random Block Read		27H
機能	ランダムなブロック読み出し	
コール	AH＝27H	
	DS：DX←オープンされているFCBの先頭アドレス	
	CX←読み出すレコード数	
リターン	AL＝00H	読み出しは正常に行われた
	AL＝01H	ファイルの終わり(EOF)を検出した．このレコードにデータは入っていない
	AL＝02H	DTAが小さ過ぎる
	AL＝03H	読み出しの途中でEOFを検出した．EOFまでのデータがDTAに読み込まれ，残りの部分は00Hで埋められる
	CX←読み出されたレコード数	

FCB内の相対レコードで指定されたレコードから，CXレジスタで指定されたレコード数がDTAに読み出されます．そして，読み出されたレコード数がCXレジスタに返されます．

このファンクションを実行したあとに，カレント・ブロックやカレント・レコードおよび相対レコードの各フィールドは，読み出した次のレコードを指すようにセットされます．

Random Block Write		28H
機能	ランダムなブロック書き込み	
コール	AH＝28H	
	DS：DX←オープンされているFCBの先頭アドレス	
	CX←書き込むレコード数（0：ファイル・サイズ・フィールドの設定）	
リターン	AL＝00H　書き込みが正常に行われた	
	AL＝01H　ディスクの空き領域がない	
	AL＝02H　DTAが小さ過ぎる	
	CX←書き込まれたレコード数	

DTAからFCB内の相対レコードで指定されたレコードへ，CXレジスタで指定されたレコード数が書き込まれます．このあと，書き込まれたレコード数がCXレジスタに返されます．

カレント・ブロックやカレント・レコードおよび相対レコードの各フィールドは，このファンクションの実行後，書き込んだ次のレコードを指すようにセットされます．

なお，CXレジスタに0を設定してこのファンクションを実行すると，ファイルの書き込みは行われずに，ファイル・サイズ・フィールドの設定のみが行われます．

〔表6-14〕ファイル名解析の制御ビット

ビット	データ	機　能
0	0	ファイル分離記号を検出した場合，すべての解析を停止する
	1	先行する分離記号は無視される
1	0	文字列にドライブ番号が入っていない場合，FCB内のドライブ番号は00H(カレント)にセットされる
	1	文字列にドライブ番号が入っていない場合，FCB内のドライブ番号は変更されない
2	0	文字列にファイル名が入っていない場合，FCB内のファイル名に8個のスペースがセットされる
	1	文字列にファイル名が入っていない場合，FCB内のファイル名は変更されない
3	0	文字列に拡張子が入っていない場合，FCB内の拡張子に3個のスペースがセットされる
	1	文字列に拡張子が入っていない場合，FCB内の拡張子は変更されない

Parse File Name		29H
機能	ファイル名の解析	
コール	AH＝29H	
	AL←解析の制御	
	DS：SI←解析する文字列の先頭アドレス	
	ES：DI←オープンされていないFCBの先頭アドレス	
リターン	AL＝00H　ワイルド・カードは使用されていない	
	AL＝01H　ワイルド・カードが使用されている	
	AL＝FFH　ドライブ名が無効	
	DS：SI←解析された文字列の中のファイル名の直後のアドレス	

DS：SIレジスタで指定されたファイル名を，ES：DIレジスタで指定されたオープンされていないFCBにセットします．このファンクションは，キーボードやコマンド・ラインから渡されたファイル名をセットして，ファイルのオープンを行う場合などに使用されます．

ALレジスタのビット0～ビット3の内容により，**表6-14**のように解析処理の方法を指定します．この場合，ビット4～ビット7は無視されます．

6-10 階層ディレクトリ管理に関するファンクション

これらのファンクションは，ver.2.11以降の階層ディレクトリのサポートにともなって用意されたファンクションで，**表6-15**のように4種類が用意されています．

〔表6-15〕ディレクトリ管理に関するファンクション

ファンクション番号	機　能	CP/M	MS-DOS ver.1.25	MS-DOS ver.2.11	MS-DOS ver.3.10	MS-DOS ver.3.30
39H	サブ・ディレクトリの作成	×	×	○	○	○
3AH	サブ・ディレクトリの削除	×	×	○	○	○
3BH	カレント・ディレクトリの変更	×	×	○	○	○
47H	カレント・ディレクトリの読み出し	×	×	○	○	○

Create Sub-Directory	39H
機能	サブ・ディレクトリの作成
コール	AH=39H DS:DX←パス名の先頭アドレス
リターン	CF=1の場合 AX←エラー・コード(表6-2) CF=0の場合 エラーなし

DS:DXレジスタが指すパス名はASCIZ文字列で表します.

Remove Directory Entry	3AH
機能	サブ・ディレクトリの削除
コール	AH=3AH DS:DX←パス名の先頭アドレス
リターン	CF=1の場合 AX←エラー・コード(表6-2) CF=0の場合 正常終了

DS:DXレジスタが指すパス名はASCIZ文字列で指定します.

Change the Current Directory	3BH
機能	カレント・ディレクトリの変更
コール	AH=3BH DS:DX←パス名の先頭アドレス
リターン	CF=1の場合 AX←エラー・コード(表6-2) CF=0の場合 正常終了

DS:DXレジスタが指すパス名はASCIZ文字列で指定します.

Return Text of Current Directory	47H
機能	カレント・ディレクトリの読み出し
コール	AH=47H DS:SI←バッファ(64バイト)の先頭アドレス DL←ドライブ番号 (00H=カレント, 01H=A, …)
リターン	CF=1の場合 AX←エラー・コード(表6-2) CF=0の場合 正常終了

このファンクションは，指定されたドライブのカレント・ディレクトリを示すパス名を，DS:SIレジスタで指定したメモリ領域にASCIZ文字列として格納します.

パス名は，そのドライブのルート・ディレクトリからの相対的な位置として返されるので，ドライブ名やルート・ディレクトリを示す最初の"¥"は含まれません.

6-11 ファイル・ハンドルに関するファンクション

これらのファンクションは，ver.2.11になってからファイル・ハンドルを用いてファイル/デバイスをアクセスするため拡張されたファンクションで，表6-16に示すようにサブファンクションを含めて20種類が用意されています.

Create a File	3CH
機能	ファイルの作成
コール	AH=3CH DS:DX←パス名の先頭アドレス CX←ファイルの属性
リターン	CF=1の場合 AX←エラー・コード(表6-2) CF=0の場合 AX←ファイル・ハンドル

DS:DXレジスタで指定されたアドレスにセットされたASCIZ文字列のファイルが，CXレジスタで指定された属性(112ページ)で新しく作成されます.

ファイルがすでに存在している場合は，既存のファイルの内容は失われ，大きさが0バイトのファイルになります．また，ファイル・ポインタはファイルの先頭にセットされます.

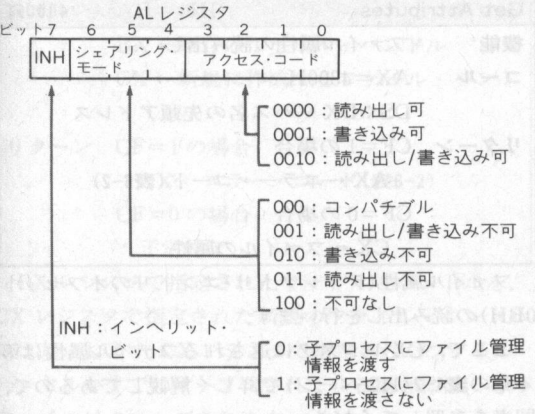
Open a File	3DH
機能	ファイルのオープン
コール	AH=3DH AL←ファイル アクセス制御コード DS:DX←パス名の先頭アドレス
リターン	CF=1の場合 AX←エラー・コード(表6-2) CF=0の場合 AX←ファイル・ハンドル

ファイルをオープンしてファイル・ハンドルを返します．以後，このオープンされたファイルは返されたファイル・ハンドルを用いてアクセスすることができます.

DS:DXレジスタにはASCIZ文字列で表されたオープンすべきファイル名の先頭アドレスをセットします．ALレジスタで設定する制御コードは，図6-5に示

〔表6-16〕ファイル・ハンドルを用いたファイル/デバイスのアクセスに関するファンクション

ファンクション番号	分類	機能	CP/M	MS-DOS ver.1.25	MS-DOS ver.2.11	MS-DOS ver.3.10	MS-DOS ver.3.30
3CH	ファイル名(ディレクトリ・エントリ)	ファイルの作成(更新あり)	×	×	○	○	○
5AH		中間ファイルの作成	×	×	×	○	○
5BH		ファイルの作成(新規のみ)	×	×	×	○	○
56H		ディレクトリ・エントリの移動	×	×	○	○	○
41H		ファイルの削除	×	×	○	○	○
4EH	検索	最初のファイルの検索	×	×	○	○	○
4FH		次のファイルの検索	×	×	○	○	○
5700H	タイム・スタンプ	タイム・スタンプの読み出し	×	×	×	○	○
5701H		タイム・スタンプの設定	×	×	×	○	○
5C00H	ロック	ファイルのロック	×	×	×	○	○
5C01H		ファイルのロック解除	×	×	×	○	○
4300H	属性	ファイル属性の読み出し	×	×	○	○	○
4301H		ファイル属性の設定	×	×	○	○	○
3DH	データ・アクセス	ファイルのオープン	×	×	○	○	○
3EH		ファイルのクローズ	×	×	○	○	○
3FH		ファイルの読み出し	×	×	○	○	○
40H		ファイルへの書き込み	×	×	○	○	○
42H		ファイル・ポインタの移動	×	×	○	○	○
45H	ハンドル	ファイル・ハンドルのコピー	×	×	○	○	○
46H		指定したファイル・ハンドルへのコピー	×	×	○	○	○

〔図6-5〕ファイル・アクセス・コントロール(3DH)

したように定義されています.

Close a File Handle	3EH
機能	ファイルのクローズ
コール	AH＝3EH BX←ファイル・ハンドル
リターン	CF＝1の場合 AX←エラー・コード(表6-2) CF＝0の場合 正常終了

ファイル・ハンドルで指定されたファイルがクローズされます. ファイルに書き込みを行った場合は, 更新されたディレクトリやFATをディスクに書き込みます.

Read from File/Device	3FH
機能	ファイル/デバイスの読み出し
コール	AH＝3FH DS：DX←入力バッファの先頭アドレス CX←読み込むべきバイト数 BX←ファイル・ハンドル
リターン	CF＝1の場合 AX←エラー・コード(表6-2) CF＝0の場合 AX←読み込まれたバイト数

BXレジスタで指定されたファイル/デバイスから, CXレジスタで指定されたバイト数のデータを, DS：DXレジスタで指定されたメモリに読み込みます.

DS：DXで指定するバッファは, FCBによるファイル・アクセスとは異なり, セグメントの境界に跨っていても連続したメモリ領域に正しく読み込まれます.

AXレジスタには, 実際に読み込まれたバイト数が返されます.

Write to File/Device		40H
機能	ファイル/デバイスへの書き込み	
コール	AH=40H DS:DX←バッファの先頭アドレス CX←書き込むべきバイト数 BX←ファイル・ハンドル	
リターン	CF=1の場合 AX←エラー・コード(**表6-2**) CF=0の場合 AX←書き込まれたバイト数	

DS:DXレジスタで指定されたメモリから，CXレジスタで指定されたバイト数のデータを，BXレジスタで指定されたファイル/デバイスに書き込みます．

DS:DXレジスタで指定するバッファは，ファンクション3FHと同様に，セグメントの境界に跨っていても連続したメモリ領域のデータが正しく書き込まれます．

Delete Directory Entry		41H
機能	ファイルの削除	
コール	AH=41H DS:DX←パス名の先頭アドレス	
リターン	CF=1の場合 AX←エラー・コード(**表6-2**) CF=0の場合 正常終了	

DS:DXレジスタで指定されたアドレスにあるASCIZ文字列のファイルの削除を行いますが，このファイルがオープンされたままでは削除されません．また，ワイルド・カードも使用できません．

このファンクションで削除できるのはファイルのディレクトリ・エントリであり，サブ・ディレクトリのディレクトリ・エントリは削除できません．サブ・ディレクトリの削除を行うには，ファンクション3AH(170ページ)を用いて行います．ただし，ファンクション3AHによってサブ・ディレクトリを削除しようとするとき，そのサブ・ディレクトリは空になっていなければなりません．

Move a File Pointer		42H
機能	ファイル・ポインタの移動	
コール	AH=42H CX:DX←移動するバイト数(32ビット符号付き) AL=00H　ポインタはファイルの先頭からオフセット(CX:DX)の位置に移動する AL=01H　ポインタは現在位置からオフセットを加算した位置に移動する AL=02H　ポインタはファイルの終わりにオフセットを加算した位置に移動する BX←ファイル・ハンドル	
リターン	CF=1の場合 AX←エラー・コード(**表6-2**) CF=0の場合 DX:AX←移動した後のファイル・ポインタの位置(先頭からのオフセット)	

このファンクションによりファイル・ポインタを自由に移動することができるので，ファイルのランダムなアクセスも可能になります．

Get Attributes		4300H
機能	ファイル属性の読み出し	
コール	AX=4300H DS:DX←パス名の先頭アドレス	
リターン	CF=1の場合 AX←エラー・コード(**表6-2**) CF=0の場合 CX←ファイルの属性	

ファイル属性(ディレクトリ・エントリのオフセット0BH)の読み出しを行います．

ここで，CXレジスタに返されるファイル属性は第4章の**表4-2**(112ページ)で詳しく解説してあるので，同表を参照してください．

〔図6-6〕DTA内のフォーマット

DTA	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	属性	ドライブ番号	ファイル名								拡張子			リザーブ		

10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
リザーブ					属性	時刻		日付		ファイルの大きさ				パック	

20	21	22	23	24	25	26	27	28	29	2A
ネーム(ASCIZ文字列)										

【注意】
1）オフセット00Hの属性は検索しようとするファイルの属性
2）オフセット15Hの属性は検索されたファイルの属性
3）パック・ネームとはスペース・コードを除いたもの

Change Attributes	4301H
機能	ファイル属性の変更
コール	AX＝4301H DS：DX ←パス名の先頭アドレス CX ←ファイルの新しい属性
リターン	CF＝1の場合 AX ←エラー・コード(表6-2) CF＝0の場合(正常終了) CX ←ファイルの属性

ファイルの属性の変更を行います.

Duplicate a File Handle	45H
機能	ファイル・ハンドルの二重化
コール	AH＝45H BX ←ファイル・ハンドル
リターン	CF＝1の場合 AX ←エラー・コード(表6-2) CF＝0の場合 AX ←新規のファイル・ハンドル

すでにオープンされているファイル・ハンドル(BXレジスタで指定)と同じファイルを扱うために,新規のファイル・ハンドルをAXレジスタに返します.

Force a Duplicate of a Handle	46H
機能	指定したファイル・ハンドルへのコピー
コール	AH＝46H BX ←既存のファイル・ハンドル CX ←新規に作成するファイル・ハンドル
リターン	CF＝1の場合 AX ←エラー・コード(表6-2) CF＝0の場合 正常終了

BXレジスタで指定されたファイル・ハンドルを,CXレジスタで指定された新規のファイル・ハンドルにコピーします.CXレジスタで指定されたファイル・ハンドルがすでにオープンされている場合には，そのファイルはクローズされます.

Find Match File	4EH
機能	一致するファイル名の検索
コール	AH＝4EH DS：DX ←パス名の先頭アドレス CX ←ファイルの属性
リターン	CF＝1の場合 AX ←エラー・コード(表6-2) CF＝0の場合 正常終了

指定されたファイル名(ワイルド・カードも可)および属性に該当する最初のファイルを検索し，現在のDTA内に図6-6のフォーマットによりその情報がセットされます.

Step Through a Directory Matching Files	4FH
機能	次に一致するファイルの検索
コール	AH＝4FH
リターン	CF＝1の場合 AX ←エラー・コード(表6-2) CF＝0の場合 正常終了

ディレクトリ内で次に一致するファイルを検索し，ファンクション4EHと同様の情報をDTAに返します.

Move a Directory Entry	56H
機能	ディレクトリ・エントリの移動
コール	AH＝56H DS：DX ←既存のファイルのパス名の先頭アドレス ES：DI ←新規のファイルのパス名の先頭アドレス
リターン	CF＝1の場合 AX ←エラー・コード(表6-2) CF＝0の場合 正常終了

このファンクションは，異なるディレクトリ間でファイル名の移動を行います.また，ファイル名を変えることにより，同一のディレクトリ内でのRENAMEの役割も行います．ファイル名にはワイルド・カードは使用できません.

このファンクションは，同一ドライブ内ディレクトリ・エントリの移動(ディレクトリが異なってもよい)を行うためのものです．これはファイル名の変更の機能を含んでいます(同一ディレクトリ内での移動の場合).

Get Date/Time of File	5700H
機能	ファイルの日付/時刻の読み出し
コール	AX＝5700H BX ←ファイル・ハンドル
リターン	CF＝1の場合 AX ←エラー・コード(表6-2) CF＝0の場合 CX ←ファイルの時刻 DX ←ファイルの日付

このファンクションでは，ファイルが最後に書き込

まれた日付/時刻の情報を読み出します．なお，これらの日付/時刻のフォーマットについては第4章(**図4-5**；113ページ)で解説してあります．

Set Date/Time of File		**5701H**
機能	ファイルの日付/時刻の変更	
コール	AX＝5701H BX←ファイル・ハンドル CX←変更する時刻 DX←変更する日付	
リターン	CF＝1の場合 AX←エラー・コード(**表6-2**) CF＝0の場合 正常終了	

このファンクションでは，ファイルが最後に書き込まれた日付や時刻の変更を行います．ただし，ファイルのクローズを行わないと，新たに設定された日付/時刻はディレクトリ・エントリには書き込まれません．日付/時刻のフォーマットについては第3章で解説してあります．

Create Temporary File		**5AH**
機能	テンポラリ・ファイルの作成	
コール	AH＝5AH CX←作成するファイルの属性 DS：DX←パス名とそれに続くメモリ領域(13バイト)へのポインタ	
リターン	CF＝1の場合 AX←エラー・コード(**表6-2**) CF＝0の場合 AX←ファイル・ハンドル	

このファンクションを発行すると，MS-DOSはCXレジスタに指定された属性でテンポラリ・ファイル(ほかのファイルと衝突しないようなファイル名をもつ)を作成し，それをアクセスするためのファイル・ハンドルを返します(読み書き両用，コンパチブル・モード)．作成されたファイルのファイル名は，DS：DXレジスタで指定されたパスの後の13バイトの領域にASCIZ文字列で格納されます．

CXレジスタに設定するファイルの属性(アトリビュート)については，第4章の**表4-2**(112ページ)に示してあります．

このファンクションで作成されたテンポラリ・ファイルは，これを作成したプロセスが終了しても自動的に消去されることはありません．

Create New File		**5BH**
機能	新しいファイルの作成	
コール	AH＝5BH CX←作成するファイルの属性 DS：DX←パス名へのポインタ	
リターン	CF＝1の場合 AX←エラー・コード(**表6-2**) CF＝0の場合 AX←ファイル・ハンドル	

このファンクションでは，DS：DXレジスタで指定されたパス名をもつファイルの作成を行います．

ファンクション3CH(ハンドルの作成：170ページ)では，指定されたパス名と同名のファイルが存在した場合は，既存のファイルの内容を0バイトとして作成します．それに対してこのファンクションでは，指定したファイル名がすでに存在している場合にはエラー(コード：50H)が返される点が異なります．

Lock		**5C00H**
機能	ファイルのロック	
コール	AX＝5C00H BX←ファイル・ハンドル CX：DX←ロックする領域のオフセット(CXが上位ワード) SI：DI←ロックする領域の大きさ(SIが上位ワード)	
リターン	CF＝1の場合 AX←エラー・コード(**表6-2**) CF＝0の場合 エラーなし	

このファンクションでは，ファイル・ハンドルで指定されたファイルの指定された範囲をロックします．ただし，share.exeが常駐していない状態では機能しません．

ロックされた領域への他のプロセスからのアクセスに対して，MS-DOSは3回(ファンクション440BHによって変更可能：177ページ)の再試行を行い，それでも拒否された場合はINT 24Hのエラー・ハンドラに制御を移します．

Unlock	5C01H
機能	ファイルのロック解除
コール	AX＝5C01H BX←ファイル・ハンドル CX：DX←ロックを解除する領域のオフセット (CXが上位ワード) SI：DI←ロックを解除する領域の大きさ(SIが上位ワード)
リターン	CF＝1の場合 AX←エラー・コード(**表6-2**) CF＝0の場合 エラーなし

このファンクションを実行すると，ファイル・ハンドルで指定されたファイルの指定された範囲(ロック時と同じでなければならない)のロックを解除します．ロックは，このファンクションを実行するかファイルをクローズしない限り解除されません．

6-12 デバイス制御に関するファンクション

デバイス(ドライバ)から制御データを取得したり，逆にデバイスに制御データを送ったりするためのファンクションは，**表6-17**のようにサブファンクションを含めて14種類が用意されています．

これらのファンクションのうち，ファンクション4408Hと440BHはMS-DOS ver.3.10になって追加され，ファンクション440CH〜440FHはMS-DOS ver.3.30で追加されました．

Get I/O Control Data	4400H
機能	デバイス・データの取得
コール	AX＝4400H BX←ファイル・ハンドル
リターン	CF＝1の場合 AX←エラー・コード(**表6-2**) CF＝0の場合 DX←デバイス情報

オープンされているファイル・ハンドルに関連したデバイスの情報を読み出します．DXレジスタに返されるデバイス・データの各ビットの機能は**図6-7**(次ページ)のように定義されています．

Set IOCTL Data	4401H
機能	デバイス・データの設定
コール	AX＝4401H BX←ファイル・ハンドル DX←デバイス・データ (ただしDH＝00H)
リターン	CF＝1の場合 AX←エラー・コード(**表6-2**) CF＝0の場合 DX←デバイス・データ

このファンクションは，ファイル・ハンドルによって指定されたデバイス(ファイル・ハンドルがファイルの場合は，そのファイルのあるデバイス)のデバイス・データを変更します．デバイス・データの機能はファンクション4400Hと同様に，**図6-7**のようになっています．

〔表6-17〕デバイス制御に関するファンクション

ファンクション番号	機能	CP/M	MS-DOS ver.1.25	MS-DOS ver.2.11	MS-DOS ver.3.10	MS-DOS ver.3.30
4400H	デバイス・データの取得	×	×	○	○	○
4401H	デバイス・データの設定	×	×	○	○	○
4402H	IOCTLデータの送出(キャラクタ・デバイス)	×	×	○	○	○
4403H	IOCTLデータの取得(キャラクタ・デバイス)	×	×	○	○	○
4404H	IOCTLデータの送出(ブロック・デバイス)	×	×	○	○	○
4405H	IOCTLデータの取得(ブロック・デバイス)	×	×	○	○	○
4406H	入力ステータスのチェック	×	×	○	○	○
4407H	出力ステータスのチェック	×	×	○	○	○
4408H	メディア交換性のチェック	×	×	×	○	○
440BH	再試行の回数と待ち時間の設定	×	×	×	○	○
440CH	一般IOCTL(ハンドル用)	×	×	×	×	○
440DH	一般IOCTL(ブロック・デバイス用)	×	×	×	×	○
440EH	論理ドライブ・マップの取得	×	×	×	×	○
440FH	論理ドライブ・マップの設定	×	×	×	×	○

〔図6-7〕DX レジスタに返されるデバイス情報

ビット	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	リザーブ	CTRL	リザーブ						ISDEV	EOF	RAW	SPECL	ISCLK	ISNUL	ISCOUT	ISCIN
	DH レジスタ								DL レジスタ							

ビット	情報名	データ	内容
0	ISCIN	1	コンソール入力装置
1	ISCOUT	1	コンソール出力装置
2	ISNUL	1	NUL 装置
3	ISCLK	1	クロック装置
4	SPECL	1	特殊な装置
5	RAW	0	この装置が動作した
		1	この装置が初期状態モードである
6	EOF	0	EOF を入力する
		1	EOF を入力しない
7	ISDEV	0	このチャネルはディスク・ファイルである (0～5 ビット目：ドライブ番号 00H＝A, 01H＝B, …)
		1	このチャネルはデバイスである
14	CTRL	0	機能コード 02H, 03H が使えない
		1	機能コード 02H, 03H が使える

Receive IOCTL Character　4402H

機能	IOCTL(I/O Control)データの受け取り (キャラクタ・デバイス)
コール	AX＝4402H BX←ファイル・ハンドル CX←受け取るIOCTLデータのバイト数 DS：DX←IOCTLデータ列の入るバッファへのポインタ
リターン	CF＝1の場合 AX←エラー・コード(表6-2) CF＝0の場合 AX←受け取ったIOCTLデータのバイト数

このファンクションは，ファイル・ハンドルによって指定されたキャラクタ・デバイスから，IOCTLデータを受け取り(読み出し)，指定されたバッファに格納します．デバイスは，IOCTLデータの送受をサポートしているデバイスでなければなりません．

すなわち，BXレジスタに指定するファイル・ハンドルはプリンタやシリアル・ポートなどのキャラクタ・デバイスのハンドルでなければなりません．

このファンクションが実行されると，AXレジスタには転送されたIOCTLデータのバイト数が返されます．このファンクションは，IOCTLインターフェースをサポートしているデバイス・ドライバに対して発行します．

Send IOCTL Character　4403H

機能	IOCTLデータの送出 (キャラクタ・デバイス)
コール	AX＝4403H BX←ファイル・ハンドル CX←送出するIOCTLデータのバイト数 DS：DX←IOCTLデータ列へのポインタ
リターン	CF＝1の場合 AX←エラー・コード(表6-2) CF＝0の場合 AX←送出されたIOCTLデータのバイト数

このファンクションは，ファイル・ハンドルによって指定されたキャラクタ・デバイスに対して，IOCTLデータを送出(書き込み)します．デバイスは，IOCTLデータの送受をサポートしているデバイスでなければなりません．

Receive IOCTL Block　4404H

機能	IOCTLデータの受け取り (ブロック・デバイス)
コール	AX＝4404H BL←ドライブ番号 (00H＝カレント，01H＝A，…) CX←受け取るIOCTLデータのバイト数 DS：DX←IOCTLデータ列の入るバッファへのポインタ
リターン	CF＝1の場合 AX←エラー・コード(表6-2) CF＝0の場合 AX←受け取ったIOCTLデータのバイト数

このファンクションは，ドライブ番号よって指定されたブロック・デバイスからIOCTLデータを受け取り(読み出し)，指定されたバッファに格納します．デバイスは，IOCTLデータの送受をサポートしているデバイスでなければなりません．デバイス・ドライバがIOCTLインターフェースをサポートしているかどうかはファンクション4400Hを実行し，得られたデバイス・データのビット14により知ることができます．

このファンクションが実行されると，AXレジスタには転送されたIOCTLデータのバイト数が返されます．

Send IOCTL Block	4405H
機能	IOCTLデータの送出 (ブロック・デバイス)
コール	AX=4405H BL←ドライブ番号 (00H=カレント, 01H=A, …) CX←送出するIOCTLデータのバイト数 DS:DX←IOCTLデータ列へのポインタ
リターン	CF=1の場合 AX←エラー・コード(表6-2) CF=0の場合 AX←送出されたIOCTLデータのバイト数

このファンクションは, ドライブ番号によって指定されたブロック・デバイスに対して, IOCTLデータを送出(書き込み)します. デバイスは, IOCTLデータの送受をサポートしているデバイスでなければなりません.

Get Input IOCTL Status	4406H
機能	入力ステータスのチェック
コール	AX=4406H BX←ファイル・ハンドル
リターン	CF=1の場合 AX←エラー・コード(表6-2) CF=0の場合 AL=00H:入力が行えない状態 (EOF検出) AL=FFH:入力が行える状態

このファンクションは, ファイル・ハンドルによって指定されたファイル/デバイスからデータの入力が可能かどうかを返します. ファイル・ポインタがEOFに達していてデータの入力ができない場合は, ALレジスタに00Hが返されます.

Get Output IOCTL Status	4407H
機能	出力ステータスのチェック
コール	AX=4407H BX←ファイル・ハンドル
リターン	CF=1の場合 AX←エラー・コード(表6-2) CF=0の場合 AL=00H:出力が行えない状態 AL=FFH:出力が行える状態

このファンクションは, ファイル・ハンドルによって指定されたファイル/デバイスに対してデータの出力が可能かどうかを返します. ただし, ファイルの場合には, たとえディスクが一杯であっても, 常にALレジスタにはFFH(出力可能)が返されます.

IOCTL is Changeable	4408H
機能	メディアの交換可能性のチェック
コール	AX=4408H BL←ドライブ番号 (00H=カレント, 01H=A, …)
リターン	CF=1の場合 AX←エラー・コード(表6-2) CF=0の場合 AX=00H:交換可能 AX=01H:交換不可能

このファンクションは, 指定されたドライブが, メディア交換可能なドライブであるかどうかを返します. たとえば, ハード・ディスクやRAMディスクでは交換不可能が返されます.

IOCTL Retry	440BH
機能	再試行の回数と待ち時間の設定
コール	AX=440BH BX←再試行の回数 CX←待ち時間 (CPUのクロックなどに依存) DX←リトライ回数
リターン	CF=1の場合 AX←エラー・コード(表6-2) CF=0の場合 正常終了

このファンクションは, MS-DOSがディスク・アクセスに失敗した際に再試行する回数と再試行の間隔(待ち時間)を設定します.

Generic IOCTL(for handles)	440CH
機能	一般IOCTL(ハンドル用)
コール	AX=440CH BX←ハンドル CH=05H:カテゴリ・コード (プリンタ・デバイス) CL←機能(マイナ)コード DS:DX←バッファへのポインタ
リターン	CF=1の場合 AX←エラー・コード(表6-2) CF=0の場合 正常終了

このファンクションは, PRINT TILL BUSYがサポートされているプリンタ・ドライバに対して, プリ

ンタへの出力の繰り返し回数を設定したり取得したりします.

機能コード(CL レジスタ)が 45H の場合には，プリンタに対する繰り返し回数の設定を行います．機能コードが 65H の場合には繰り返し回数の取得を行います．

DS：DX レジスタは，繰り返し回数が格納されているワードへのポインタであり，このワードはデバイス・ドライバがデバイスの BUSY を待つ回数を表します．

Generic IOCTL(for block devices)	440DH
機能	一般 IOCTL(ブロック・デバイス用)
コール	AX＝440DH BL←デバイス番号 (00H＝デフォルト，01H＝A，…) CH＝08H：カテゴリ・コード CL←機能コード DS：DX←パラメータ・ブロック−1へのポインタ
リターン	CF＝1 の場合 AX←エラー・コード(**表6-2**) CF＝0 の場合 正常終了

このファンクションでは，ブロック・デバイスに対して，デバイス・パラメータの設定/取得などを行います．このファンクションでの機能は，CLレジスタの機能コードで指定することができ，その内容は**表6-18** のように定義されています．

〔表6-18〕CL レジスタで指定する機能コード

コード	機　能
40H	デバイス・パラメータの設定
60H	デバイス・パラメータの取得
41H	論理デバイス上のトラックの書き出し
61H	論理デバイス上のトラックの読み出し
42H	論理デバイス上のトラックのフォーマット
62H	論理デバイス上のトラックのベリファイ

〔図6-8〕機能コード 40H におけるパラメータ・ブロック (440DH)

DS：DX →

1バイト	特殊ファンクション
1バイト	デバイス・タイプ
1バイト	デバイス属性
1ワード	シリンダ数
1バイト	メディア・タイプ
13バイト	デバイス BPB
可変長	トラック・レイアウト

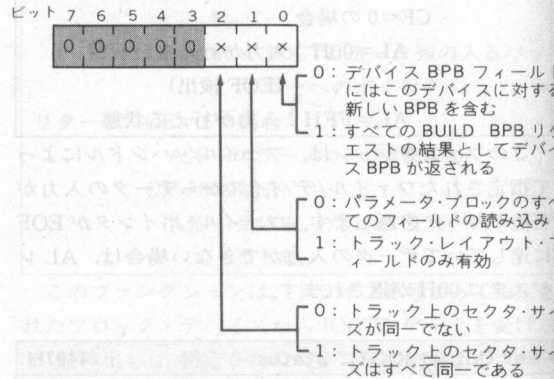
【機能コード(CL レジスタ)＝40H】

DS：DX レジスタで指すパラメータ・ブロックは，**図6-8** のフォーマットで指定します．特殊ファンクション・フィールドには**図6-9** で示したビットを設定します．デバイス・タイプには，物理デバイスのメディアの種類を**表6-19** にしたがって設定します．デバイス属性フィールドには，**図6-10** にしたがって各ビットを設定します．

シリンダ数には，物理デバイスがサポートできるシリンダ数の最大値が設定されます．メディア・タイプは，複数のメディアを使用できるドライブにおいて，どの種類のメディアがドライブにセットされているのかを表します．

デバイス BPB フィールドには，特殊ファンクション・フィールドのビット 0 が“0”で指定された場合に，デバイスの新しい BPB を設定します．もし，特殊ファンクション・フィールドのビット 0 が“1”の場合には，このフィールドにはデバイス・ドライバから BUILD BPB リクエストによって BPB が返されます．

トラック・レイアウト・フィールドは可変長テーブルであり，最初の 1 ワードにセクタの総数を指定し，次にセクタ番号とセクタ・サイズの 2 ワードを対にし

〔図6-9〕特殊ファンクション・フィールドの意味 (440DH)

〔表6-19〕デバイス・タイプ・フィールドの意味

値	意　味
0	320/360K バイト
1	(未使用)
2	640/720K バイト
3	256K バイト(8″単密度)
4	1 M バイト
5	ハード・ディスク
6	(未使用)
7	その他

て，セクタ総数だけ繰り返して指定します(図6-11).

【機能コード＝60H】

パラメータ・ブロックの各フィールドは図6-8と同じです．ただし，特殊ファンクション・フィールドはビット1のみが有効になり，ほかのビットは“0”にします．また，トラック・レイアウト・フィールドは必要としません．

【機能コード＝41H および 61H】

論理デバイス上のトラックに書き出しを行うには，CLレジスタに41Hを設定します．また，トラックの読み出しを行うには，CLレジスタに61Hを設定します．この場合に，パラメータ・ブロックのフォーマットは図6-12に示した内容になります．

特殊ファンクション・フィールドには，セクタの総数を指定します．転送アドレス・フィールドには，読み出したデータを格納するバッファのアドレス，または書き出すべきデータの入っているバッファのアドレスを設定します．

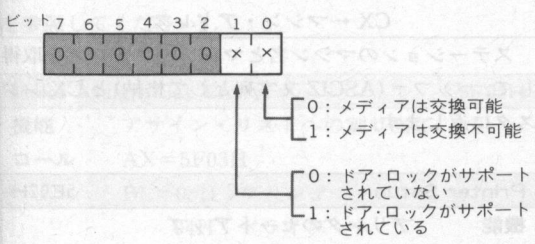
【機能コード＝42H および 62H】

論理デバイス上のトラックのフォーマットとベリファイをする場合はCLレジスタに42Hを設定します．論理デバイス上のトラックのデータのベリファイだけを行う場合は，CLレジスタに62Hを指定します．このとき，パラメータ・ブロックのフォーマットは，図6-13に示した内容になります．

特殊ファンクション・フィールドには，00Hを設定しなければなりません．ヘッド・フィールドには，フォーマットまたはベリファイを実行するヘッド番号を指定します．シリンダ・フィールドには，フォーマットまたはベリファイを実行するシリンダ番号を指定します．

Get Logical Drive Map	440EH
機能	論理ドライブ・マップの取得
コール	AX＝440EH
	BX←ドライブ番号
	(00H＝デフォルト，01H＝A，…)
リターン	CF＝1の場合
	AX←エラー・コード(表6-2)
	CF＝0の場合
	AL←物理ドライブ番号

MS-DOS ver.3.30では，1個の物理的なブロック・デバイスに対して，複数の論理ドライブのマッピングをサポートしています．このファンクションでは，論理ドライブが，現在どの物理デバイスにマッピングされているかを調べることができます．

〔図6-10〕デバイス属性フィールドの各ビットの意味(440DH)

〔図6-11〕トラック・レイアウト(440DH)

〔図6-12〕機能コード41Hおよび61Hにおけるパラメータ・ブロック

DS：DX →

1バイト	特殊ファンクション
1ワード	ヘッド
1ワード	シリンダ
1ワード	第1セクタ
1ワード	セクタ数
2ワード	転送アドレス

〔図6-13〕機能コード42Hおよび62Hにおけるパラメータ・ブロック

DS：DX →

1バイト	特殊ファンクション
1ワード	ヘッド
1ワード	シリンダ

Set Logical Drive Map	440FH
機能	論理ドライブ・マップの設定
コール	AX＝440FH BX←ドライブ番号 (00H＝デフォルト, 01H＝A, …)
リターン	CF＝1の場合 AX←エラー・コード(表6-2) CF＝0の場合 正常終了

このファンクションでは，現在物理デバイスにマップされている論理デバイスを変更します．このファンクションやファンクション440EHは，2個以上の論理デバイスを一つの物理デバイスへマップするときにたいへん便利です．

6-13 MS-Networksに関するファンクション

これらのファンクションは，MS-Networksに関連したファンクションであり，MS-Networksが起動されていないと意味がありません．

MS-Networksに関するファンクションは，サブファンクションを含めて表6-20のように7種類が用意されています．

IOCTL is Redirected Block	4409H
機能	ローカル/リモート・チェック (ドライブ)
コール	AX＝4409H BL←ドライブ番号 (00H＝デフォルト, 01H＝A, …)
リターン	CF＝1の場合 AX←エラー・コード(表6-2) CF＝0の場合 DX←デバイス・アトリビュート・ワード

指定されたドライブがローカル・ドライブ(ステーションに接続された物理的なドライブ)であれば，DXレジスタにデバイス・ヘッダのアトリビュート・ワードを返します．指定したドライブがサーバにリダイレクトされているリモート・ドライブの場合には，DXレジスタには“1000H”がセットされます．

通常のプログラムを作成する場合には，ドライブがローカルであるかリモートであるかを区別するのは透過性の点からも好ましくありません．特に必要な場合(ディスク交換を要求する場合など)以外は区別すべきではありません．

IOCTL is Redirected Handle	440AH
機能	ローカル/リモート・チェック (ファイル・ハンドル)
コール	AX＝440AH BX←ファイル・ハンドル
リターン	CF＝1の場合 AX←エラー・コード(表6-2) CF＝0の場合 DX←IOCTLビット・フィールド

指定されたファイル・ハンドルがローカル・デバイス(ステーションに接続された物理的なデバイス)であるかどうかを調べます．ファイル・ハンドルで指定されたファイルが，ローカル・ドライブ上のファイルであれば，DXレジスタのビット15に“1”を返します．

通常のプログラムを作成する場合には，ドライブがローカルであるかリモートであるかを区別するのは透過性の点からも好ましくありません．特に必要な場合(ディスク交換を要求する場合など)以外は区別すべきではありません．

Get Machine Name	5E00H
機能	マシン名の取得
コール	AX＝5E00H DS：DX←16バイトのバッファへのポインタ
リターン	CF＝1の場合 AX←エラー・コード(表6-2) CF＝0の場合 CX←マシン・アドレス

ステーションのマシン名とマシン・アドレスを取得して，バッファ(ASCIZ文字列として格納)とCXレジスタに返します．

Printer Setup	5E02H
機能	プリンタのセットアップ
コール	AX＝5E02H BX←アサイン・リスト・エントリ番号 CX←セットアップ文字列の長さ DS：SI←セットアップ文字列へのポインタ
リターン	CF＝1の場合 AX←エラー・コード(表6-2) CF＝0の場合 正常終了

ネットワーク上のプリンタに出力する際に先頭に付加する文字列を設定します．

〔表6-20〕MS-Networksに関するファンクション

ファンクション番号	機能	CP/M	MS-DOS ver.1.25	MS-DOS ver.2.11	MS-DOS ver.3.10	MS-DOS ver.3.30
4409H	ローカル/リモート・チェック(ドライブ)	×	×	×	○	○
440AH	ローカル/リモート・チェック(ハンドル)	×	×	×	○	○
5E00H	ステーションのマシン名の取得	×	×	×	○	○
5E02H	プリンタのセットアップ	×	×	×	○	○
5F02H	アサイン・リストの参照	×	×	×	○	○
5F03H	アサイン・リストへの割り当て	×	×	×	○	○
5F04H	アサイン・リストの割り当て取り消し	×	×	×	○	○

Get Assign List Entry　5F02H

機能　アサイン・リストの参照

コール　AX＝5F02H
　BX←アサイン・リスト・エントリ番号
　DS：SI←ローカル名バッファへのポインタ
　ES：DI←リモート名バッファへのポインタ

リターン　CF＝1の場合
　AX←エラー・コード(表6-2)
　CF＝0の場合
　BL＝03H：プリンタ
　　04H：ドライブ
　CX←ユーザ設定値

BXレジスタで指定されたアサイン・リストの割り当て状況を返します．詳しくはファンクション5F03Hを参照してください．

Make Assign List Entry　5F03H

機能　アサイン・リストへの割り当て

コール　AX＝5F03H
　BL＝03H：プリンタ
　　04H：ドライブ
　CX←ユーザ設定値
　DS：SI←ソース・デバイス名へのポインタ
　ES：DI←デスティネーション・デバイス名へのポインタ

リターン　CF＝1の場合
　AX←エラー・コード(表6-2)
　CF＝0の場合
　正常終了

ソース・デバイス名で指定したデバイス(プリンタ(PRN)またはドライブ(ドライブ名＞：))をデスティネーション・デバイス名で指定したサーバのデバイスにリダイレクトします．デスティネーション・デバイス名は，次のような書式で指定します(パスワードは省略可)．

￥<サーバのマシン名>￥<別名><00H>
[<パスワード>]<00H>

CXレジスタには，ファンクション5F02Hを実行した際にCXレジスタに返す値を設定します(ユーザが自由に設定可能)．

Cancel Assign List Entry　5F04H

機能　アサイン・リストの割り当て取り消し

コール　AX＝5F04H
　DS：SI←ソース・デバイス名へのポインタ

リターン　CF＝1の場合
　AX←エラー・コード(表6-2)
　CF＝0の場合
　正常終了

ソース・デバイス名で指定したアサイン・リストの割り当てを解除します．

すなわち，このファンクションでは，プリンタまたはディスク・ドライブなど，ファンクション5F03Hで作成されたネットワーク，ディレクトリへのリダイレクトをキャンセルします．DS：SIレジスタには，キャンセルするリダイレクトのプリンタまたはドライブ名(ソース・デバイス)を表すASCIZ文字列へのポインタ(アドレス)を指定します．

第7章 システム・コール応用プログラム集

CとMASMとプログラミング

前章では，MS-DOS ver.3.3のファンクション・コールについて，個々の機能と使いかたを解説しました．ここでは，これらのシステム・コールを使用したアプリケーション・プログラムの作成例を紹介していきます．

ここでも，ほとんどのプログラムはC言語とアセンブリ言語を用いて記述しています．C言語のプログラムでは，主としてプログラムの実行制御やメッセージの表示などを行う部分を記述しています．また，アセンブリ言語でのプログラムはファンクション・コールを行う部分をプロシージャとして記述し，CPUのレジスタへのパラメータ設定や，戻り値の扱いかたをわかりやすくしてあります．

プログラムは，ツールとして活用できるものを中心にしました．CとMASMとの相互関係や，メモリに常駐するTSR型プログラム，MS-DOSのファイル・アクセスの方法など，参考になると思います．

7-1 文字と文字列の操作

■ 英小文字を大文字に変換する

リスト7-1(upper.c)および**リスト7-2**(uppersub.asm)は，標準入力から入力された文字が英小文字の場合に大文字に変換して標準出力に出力するプログラムupper.exeのソース・リストです．

● upper.c

リスト7-1はプログラムupper.exeのCソース部分です．同リストにおいて関数mainでは，最初にプログラム・タイトルを表示したのち無限ループに入り，関数(サブルーチン)upperを用いて文字の入出力を行います．

〔リスト7-1〕
プログラムupper.c

```
 1: /*******************************************************************
 2: *
 3: *   機    能:   入力文字のエコーと大文字変換表示
 4: *   サ    ブ:   uppersub.asm
 5: *   生    成:   masm /ML uppersub;
 6: *               cl -J -AS upper.c uppersub
 7: *   使用方法:   upper
 8: *
 9: *******************************************************************/
10:
11: void main (void);
12: void upper (void);
13: /*******************************************************************
14: *
15: *   関数名:   main ()
16: *   機  能:   文字入出力関数 ( upper) の起動
17: *   入  力:   なし
18: *   出  力:   なし
19: *
20: *******************************************************************/
21: void main ()
22: {
23:     printf ("¥n *** 文字変換プログラム Ver.1.1 ***¥n¥n");
24:     for (; ;) {
25:         upper ();
26:     }
27:     exit (0);
28: }
```

● uppersub.asm

リスト7-2 はプログラム upper.exe のアセンブリ・ソース部分です．サブルーチン(関数)upper は，ファンクション 01H を用いて文字入力を行い，入力された文字が英小文字の範囲であるかどうかを調べ，もし英小文字の場合は英大文字に変換します．そして，ファンクション 02H を用いてその文字を標準出力に出力します．このプログラム upper.exe は，ctrl-C の入力によってプログラムを終了します．

◆ 生成方法

プログラム upper.exe は，次の手順で分割アセンブル/コンパイルして作成します．

```
masm /ML uppersub;
cl -J -As upper.c uppersub
```

◆ 実行サンプル

リスト7-3 はプログラム upper.exe の実行例を示しています．

① まず，プログラム upper.exe を起動する．

② キーボードから入力した英小文字が大文字に変換されて表示される．

③ 英字以外の数字や記号などの文字は変換されない．

④ プログラムの終了は ctrl-C の入力により行う．

ASCII コードを表示する

リスト7-4(code.c)，**リスト7-5**(codesub.asm)は，標準入力から入力された文字に対し，その ASCII コードと表示可能な文字(1バイト)であればその文字の表示を行うプログラム code.exe のソース・リストです．

〔リスト7-2〕
プログラム upper.asm

```
 1: ;*************************************************************************
 2: ;
 3: ;    機      能:    入力された文字を大文字に変換する
 4: ;    ファンクション :    01H(キーボード入力)
 5: ;                   02H(文字の出力)
 6: ;    生      成:    masm /ML uppersub;
 7: ;
 8: ;*************************************************************************
 9:                .MODEL   SMALL, C
10:                .CODE
11: ;*************************************************************************
12: ;
13: ;      ルーチン名 :    toupper
14: ;      機    能 :    大文字変換
15: ;      func    :    01H(キーボード入力)
16: ;                   02H(文字の出力)
17: ;      入    力:    なし
18: ;      出    力:    なし
19: ;
20: ;*************************************************************************
21: upper          PROC
22:                mov      ah, 01h              ;ファンクション 01H
23:                int      21h
24:                cmp      al, 'z'
25:                ja       upper
26:                cmp      al, 'a'
27:                jb       upper
28:                sub      al, 'a' - 'A'
29:                mov      dl, al
30:                mov      ah, 02h              ;ファンクション 02H
31:                int      21h
32: isupper:
33:                ret
34: upper          ENDP
35:                END
```

〔リスト7-3〕
プログラム upper.exe の実行例

```
R>upper ⏎…プログラムの起動①

*** 文字変換プログラム Ver.1.1 ***

aAbBcCdDeEfF12345xXyYzZ,,/¥^C ← ctrl-C でプログラム終了④
R>
      ↑                  ↑         ↑
                                英字以外は変換されない③
英小文字が英大文字に変換される②
```

● code.c

リスト7-4はプログラムcode.exeのCソース部分です．同リストにおいて関数mainでは，最初にプログラム・タイトルを表示したのち無限ループに入ります．

そして，関数(サブルーチン)func_08を用いてエコーバックなしの文字入力(ファンクション08H)を行います．

関数func_08からは，戻り値として入力された文字

〔リスト7-4〕
プログラムcode.c

```
 1: /**********************************************************************
 2: *
 3: *    機    能:   入力文字コードの表示
 4: *    サ    ブ:   codesub.asm
 5: *    生    成:   masm /ML codesub;
 6: *                cl -J -AS code.c codesub
 7: *    使用方法:   code
 8: *
 9: **********************************************************************/
10: #include        <stdio.h>
11:
12: void main (void);
13: int func_08 (void);
14: /**********************************************************************
15: *
16: *     関数名:  main ()
17: *     機  能:  文字の読み込みとASCIIコードの表示
18: *     入  力:  なし
19: *     出  力:  なし
20: *
21: **********************************************************************/
22: void main ()
23: {
24:     int c;
25:
26:     printf ("¥n *** 文字コード・プリント・プログラム Ver.1.1 ***¥n¥n");
27:     for (; ;) {
28:         c = func_08 ();
29:         if (c == 0x7F || c < ' ' || c >= 0xF8) {
30:             printf ("コード = %02X ---- 制御コードです.¥n", c);
31:         } else {
32:             printf ("コード = %02X ---- %c¥n", c & 0x0FF, c);
33:         }
34:     }
35:     exit (0);
36: }
```

〔リスト7-5〕
プログラム
codesub.asm

```
 1: ;**********************************************************************
 2: ;
 3: ;   機      能:  キー入力の取り込み
 4: ;   ファンクション:  08H (エコーなしキーボード入力)
 5: ;   生      成:  masm /ML codesub;
 6: ;
 7: ;**********************************************************************
 8:                 .MODEL  SMALL, C
 9:                 .CODE
10: ;**********************************************************************
11: ;
12: ;   ルーチン名:  func_08
13: ;   機    能:  キー入力
14: ;   func    :  08H (エコーなしキーボード入力)
15: ;   入    力:  なし
16: ;   出    力:  AX ··· 文字コード
17: ;
18: ;**********************************************************************
19: func_08         PROC
20:                 mov         ah, 08h             ;ファンクション08H
21:                 int         21h                 ;キー入力
22:                 xor         ah, ah
23:                 ret
24: func_08         ENDP
25:                 END
```

コードが返されるので，そのASCIIコードと文字の表示を行います．この際に制御コードなど表示の不可能なコードについては，ASCIIコードの表示だけを行います．

● codesub.asm

リスト7-5 はプログラムcode.exeのアセンブリ・ソース部分です．サブルーチン(関数)func_08は，ファンクション08Hを用いて文字入力を行い，その文字コードをAXレジスタに格納して返します．

◆ 生成方法

プログラムcode.exeは，次の手順で分割アセンブル/コンパイルして作成します．

```
masm /ML codesub;
cl -J -As code.c codesub
```

◆ 実行サンプル

リスト7-6 はプログラムcode.exeの実行例を示しています．

① まず，プログラムcode.exeを起動する．
② 通常の(表示可能な)文字の入力．
③ ファンクション・キー(F1〜F3)の入力．ファンクション・キーにはesc-S，esc-T，esc-Uの順でコードが割り当てられていることがわかる．
④ ESCキーのみの入力．
⑤ ctrl-A，ctrl-Bなどの入力．
⑥ カーソル・キーの入力．
⑦ リターン・キーの入力．
⑧ プログラムを終了するにはctrl-Cを入力する．

文字列を入出力する

リスト7-7 (gstr.c)および**リスト7-8** (gstrsub.asm)は，ファンクション0AHを用いて入力した文字列を，ファンクション09Hを用いて出力するプログラムgstr.exeのソース・リストです．

● gstr.c

リスト7-7 はプログラムgstr.exeのCソース部分です．同リストにおいて，関数mainでは最初にプログラム・タイトルを表示したのち，forループによって無限ループに入ります．forループでは，関数(サブルーチン)func_0aを用いて最大255までの文字をバッファstrに入力します．ここで，関数func_0aからは入力文字数が返されるので，もし文字数が0(CRのみ)の場合は，forループから脱出してプログラムgstr.exeを終了します．

文字列の入力が終わったら，関数get_strから返された文字数の表示を行います．そして，文字列バッファstrにライブラリ関数strcatを用いてターミネータである"$"を連結します．これらの処理が終わったら，関数func_09を用いて文字列の出力を行います．

ここで，文字列バッファは図6-1(157ページ)に示したように，その1バイト目には入力すべき文字数(1〜255)が，2バイト目には実際に入力された文字数が

〔リスト7-6〕プログラムcode.exeの実行例

```
R>code ⏎… プログラムの起動①

 *** 文字コード・プリント・プログラム Ver.1.1 ***

コード = 61 ---- a                    ┐
コード = 62 ---- b                    │ 通常の文字コード②
コード = 63 ---- c                    │
コード = 64 ---- d                    ┘
コード = 1B ---- 制御コードです．  ┐ F1 ┐
コード = 53 ---- S                 ┘    │
コード = 1B ---- 制御コードです．  ┐ F2 │ ファンクション・キー③
コード = 54 ---- T                 ┘    │
コード = 1B ---- 制御コードです．  ┐ F3 │
コード = 55 ---- U                 ┘    ┘
コード = 1B ---- 制御コードです．  … ESCキーの入力④
コード = 01 ---- 制御コードです．  ┐ ctrlキーの入力⑤
コード = 02 ---- 制御コードです．  ┘
コード = 0B ---- 制御コードです．  ┐
コード = 0C ---- 制御コードです．  │ カーソル・キーの入力⑥
コード = 08 ---- 制御コードです．  ┘
コード = 0A ---- 制御コードです．  ┐ リターン・キーの入力⑦
コード = 0D ---- 制御コードです．  ┘
^C … ctrl-Cにより終了⑧
R>
```

格納され，実際の文字列は3バイト目以降に格納されていることに注意が必要です．

● gstrsub.asm

リスト7-8はプログラムgstr.exeのアセンブリ・ソース部分です．サブルーチン(関数)func_0aはファンクション0AHを用いて，引数arg2で指定された文字列バッファに，引数arg1で指定された(最大の)文字数を入力して格納します．

ここでは，ファンクション0AHの実行に先立って，最大文字数(arg1)をバッファの1バイト目に格納しています．文字列の入力が終わったら，その文字列の最後に00Hを格納してASCIZ文字列にしておきます．

サブルーチンfunc_09は，引数arg1で渡されたバッファの内容を，ファンクション09Hを用いて標準出力に出力します．ファンクション09Hでは，文字列の最後には"$"が格納されていなければなりません．

◆ 生成方法

プログラムgstr.exeは，以下の手順で分割アセンブル/コンパイルして作成します．

masm /ML gstrsub;

cl −J −As gstr.c gstrsub

◆ 実行サンプル

リスト7-9はプログラムgstr.exeの実行例を示しています．

① まず，プログラムgstr.exeを起動する．

② プログラム・タイトルを表示した後に文字入力を促してくるので適当な文字を入力する．

③ 入力された文字数が表示される．

④ 漢字(全角文字)を入力してみる．

⑤ 当然のことながら，文字数は2倍されて表示される．

⑥ キャリッジ・リターンのみでプログラムgstr.exeを終了する．

〔リスト7-7〕プログラムgstr.c

```
 1: /*****************************************************************
 2: *
 3: *  機    能 :  文字列入力 ( func 0AH) と文字出力 ( func 09H)
 4: *  サ    ブ :  gstr.asm
 5: *  生    成 :  masm /ML gstrsub;
 6: *              cl -J -AS gstr.c gstrsub
 7: *  使用方法 :  gstr
 8: *
 9: *****************************************************************/
10: #include    <stdio.h>
11: #include    <string.h>
12:
13: void main (void);
14: int  func_0a (int, char *);      /* 文字列入力 */
15: void func_09 (char *);           /* 文字列出力 */
16: /*****************************************************************
17: *
18: *    関数名 :  main ()
19: *    機  能 :  文字列入力と出力
20: *    入  力 :  なし
21: *    出  力 :  なし
22: *
23: *****************************************************************/
24: void main ()
25: {
26:     int n;
27:     char str [256];
28:
29:     printf ("¥n *** func 09H & 0AH テスト・プログラム Ver.1.1 ***¥n¥n");
30:     for (; ;){
31:         printf ("文字列を入力して下さい ( CRのみで終了 ) ¥n");
32:         if (! (n = func_0a (255, str))) {
33:             break;
34:         }
35:         printf ("¥n入力文字数 = %d¥n", n);
36:         strcat (str + 2, "$");
37:         func_09 (str + 2);
38:         printf ("¥n");
39:     }
40:     printf ("¥n");
41:     exit (0);
42: }
```

〔リスト7-8〕プログラム gstrsub.asm

```
 1: ;*************************************************************
 2: ;
 3: ;    機     能 :   文字列入出力サブルーチン
 4: ;    ファンクション :   09H（文字列出力）
 5: ;                   0AH（バッファード・コンソール入力）
 6: ;    生     成 :   masm /ML gstrsub;
 7: ;
 8: ;*************************************************************
 9:             .MODEL  SMALL, C
10:             .CODE
11: ;*************************************************************
12: ;
13: ;     ルーチン名 :  func_0a
14: ;     機    能 :  文字列の入力
15: ;     func      :  0AH（バッファード・キーボード入力）
16: ;     入    力 :  arg1 ･･･ 入力すべき文字数
17: ;                  arg2 ･･･ バッファへのポインタ
18: ;     出    力 :  AX   ･･･ 入力された文字数
19: ;
20: ;*************************************************************
21: func_0a     PROC    arg1:WORD, arg2:PTR
22:             push    bx
23:             push    si
24:             mov     ax, arg1
25:             mov     si, arg2
26:             mov     dx, si
27:             mov     [si], al                ;入力すべき文字数
28:             mov     ah, 0Ah                 ;ファンクション 0AH
29:             int     21h
30:             mov     al, [si + 1]            ;入力された文字数
31:             xor     ah, ah
32:             mov     bx, ax
33:             mov     BYTE PTR [si + bx + 2], 0
34:             pop     si
35:             pop     bx
36:             ret
37: func_0a     ENDP
38:
39: ;*************************************************************
40: ;
41: ;     ルーチン名 :  func_09
42: ;     機    能 :  文字列の出力
43: ;     func      :  09H（文字列の出力）
44: ;     入    力 :  arg1 ･･･ 文字列へのポインタ
45: ;     出    力 :  なし
46: ;
47: ;*************************************************************
48: func_09     PROC    arg1:PTR
49:             push    dx
50:             mov     dx, arg1
51:             mov     ah, 09h
52:             int     21h
53:             pop     dx
54:             ret
55: func_09     ENDP
56:             END
```

〔リスト7-9〕プログラム gstr.exe の実行例

```
R>gstr⏎… プログラムの起動①

  *** func 09H & 0AH  テスト・プログラム Ver.1.1 ***

文字列を入力して下さい（CRのみで終了）
abcdef ⏎… 適当な文字の入力②
入力文字数 = 6 … 入力された文字数の表示③
abcdef … 入力文字の表示
文字列を入力して下さい（CRのみで終了）
ＭＳ－ＤＯＳ⏎… 日本語（全角文字）の入力④
入力文字数 = 12 … 文字数は2倍になる⑤
ＭＳ－ＤＯＳ … 入力文字の表示
文字列を入力して下さい（CRのみで終了）
⏎… キャリッジ・リターンでプログラム終了⑥
R>
```

〔リスト7-10〕プログラム praux.c

```
 1: /*********************************************************************
 2: *
 3: *    機     能:   標準入力の文字をプリンタと通信回線に出力
 4: *    サ     ブ:   praux.asm
 5: *    生     成:   masm /ML prauxsub;
 6: *                 cl -J -AS praux.c prauxsub
 7: *    使用方法:    praux
 8: *
 9: *********************************************************************/
10: #include    <stdio.h>
11:
12: void main (void);
13: /*********************************************************************
14: *
15: *     関数名:  main ()
16: *     機  能:  入力文字の印字と回線への出力
17: *     入  力:  なし
18: *     出  力:  なし
19: *
20: *********************************************************************/
21: void main ()
22: {
23:     int c;
24:
25:     printf ("¥n *** プリンタ＆通信回線出力プログラム Ver.1.1 ***¥n¥n");
26:     printf ("入力文字 (ctrl-Zで終了) ¥n");
27:     while ((c = getchar ()) != EOF) {
28:         put_ch (c);
29:     }
30:     printf ("¥n");
31:     exit (0);
32: }
```

プリンタ/AUX出力を行う

リスト7-10(praux.c)，**リスト7-11**(prauxsub.asm)に示したプログラムは，標準入力から入力された文字をプリンタと補助入出力(通信回線)に出力するプログラム praux.exe のソース・リストです．

● praux.c

リスト7-10はプログラム praux.exe のCソース部分です．同リストにおいて，関数 main では最初にプログラム・タイトルを表示したのち while ループに入ります．

while ループでは，ライブラリ関数 getchar を用いて標準入力から文字を入力し，関数(サブルーチン) put_ch を用いて入力した文字をプリンタと補助入出力に出力します．ここで，入力された文字が EOF(ファイルの終端または ctrl-Z)であれば，while ループから脱出してプログラム praux.exe を終了します．

● prauxsub.asm

リスト7-11はプログラム praux.exe のアセンブリ・ソース部分です．同リストにおいて，サブルーチン(関数)put_ch は，引数 arg1 で渡された文字をファンクション 04H およびファンクション 05H を用いて，プリンタと補助入出力に出力します．

ここで，もし LF コード(0AH)に出会ったら，改行コード(0DH)も出力して正常に改行されるようにしておきます．

◆ 生成方法

プログラム praux.exe は，下記の手順で分割アセンブル/コンパイルして作成します．

masm /ML prauxsub;

cl −J −As praux.c prauxsub

◆ 実行サンプル

リスト7-12は，プログラム praux.exe の実行例を示しています．

① まず，プログラム praux.exe を起動する．

② すると，文字の入力を促してくるので適当な文字を入力する．ここで，キャリッジ・リターンの入力によってバッファの内容がプリンタや補助入出力に送られる．

③ 同様に文字入力を行う．

④ ctrl-Z を入力すると文字入力の終了となる．ここで，もし標準入力としてファイルをリダイレクトしている場合は，ファイルの終端に達すると自動的に EOF が送られて，プログラム praux は終了する．

⑤ 入力された文字がプリンタと補助出力に対して送られている．

〔リスト7-11〕
プログラム
prauxsub.asm

```
 1: ;**************************************************************
 2: ;
 3: ;   機    能 :   文字の出力（プリンタ＆通信回線）
 4: ;   ファンクション :   04H（補助出力）
 5: ;               05H（プリンタ出力）
 6: ;   生    成 :   masm /ML prauxsub;
 7: ;
 8: ;**************************************************************
 9:             .MODEL  SMALL, C
10:             .CODE
11: ;**************************************************************
12: ;
13: ;   ルーチン名 :   put_ch
14: ;   機    能 :   プリンタと通信回線への文字出力
15: ;   func     :   04H（補助出力）
16: ;               05H（プリンタ出力）
17: ;   入    力 :   arg1 ・・・ 文字コード
18: ;   出    力 :   なし
19: ;
20: ;**************************************************************
21: put_ch      PROC    arg1:WORD
22:             mov     dx, arg1            ;文字コード
23:             mov     ah, 04h
24:             int     21h                 ;ファンクション 04H
25:             mov     ah, 05h
26:             int     21h                 ;ファンクション 05H
27:             cmp     dl, 0Ah
28:             jne     exit
29:             mov     dl, 0Dh
30:             int     21h                 ;ファンクション 05H
31:             mov     ah, 04h
32:             int     21h                 ;ファンクション 04H
33: exit:
34:             ret
35: put_ch      ENDP
36:             END
```

〔リスト7-12〕プログラム praux.exe の実行例

```
R>praux ⏎ … プログラムの起動①

 *** プリンタ＆回線出力プログラム Ver.1.1 ***
入力文字（ctrl-Zで終了）
abcd ⏎ … 適当な文字の入力②
efgh ⏎ … 同様に文字を入力③
^Z   … ctrl-Z の入力でプログラム終了④

R>⏎

abcd ⎫
efgh ⎭ 入力された文字がプリンタと補助出力に送られる⑤
```

7-2 時刻・日付の操作

ストップ・ウォッチ

リスト7-13(keyin.c)と**リスト7-14**(keyinsub.asm)は，“g”または“G”キーの入力によって時間計測を開始し，任意のキー入力によって表示を停止するプログラム keyin.exe のソース・リストです．

● keyin.c

リスト7-13 はプログラム keyin.exe のＣソース部分です．同リストにおいて，構造体 _TIME は時間データの構造を定義しています．

関数 main では，最初にプログラム・タイトルを表示したのち，プログラムのスタートやストップ方法を表示し，次に構造体の各メンバを初期化しておきます．そして，関数(サブルーチン) get_time によって，キー入力と時間データの読み込みを行います．

関数 time_msg は，ポインタ ptr で渡された構造体の時間データの表示を行います．

● keyinsub.asm

リスト7-14 はプログラム keyin.exe のアセンブリ・ソース部分です．構造体 _TIME は，**リスト7-13** のＣソースの構造体 _TIME に対応するもので，時間データの格納を行うデータ領域の構造を定義します．

サブルーチン(関数) get_time は，引数 arg1 で指定されたデータ領域に時間データを格納します．まず，ファンクション 06H を用いてキーボードから文字が入力されたかどうかを調べます．もし入力された文字が“G”か“g”であれば，次にファンクション 2CH (Get Time) を用いて時間データの読み込みを行います．そして，読み込んだ時間データの秒データに変化がなけ

ればループによってファンクション 2CH を繰り返して実行します．

もし，読み込んだ時間データの秒データに変化があれば，そのデータへのポインタを引数としてサブルーチン(関数)time_msg を用いて時間データの表示を行います．

次にファンクション 0BH を用いて，どれかのキーが押されたかを調べ，もし押されていなければ再び時間データの読み込みと表示を繰り返します．もし，何かのキーが押されていれば，ファンクション 0CH を用いてキー入力の先行バッファ(タイプ・アヘッド・バッファ)を空にして，サブルーチン get_time を終了し

〔リスト7-13〕
プログラム keyin.c

```
 1: /*********************************************************************
 2: *
 3: *    機    能 :   キーの入力により時刻を表示
 4: *    サ    ブ :   keyinsub.asm
 5: *    生    成 :   masm /ML keyinsub;
 6: *                 cl -J -AS keyin.c keyinsub
 7: *    使用方法 :   keyin
 8: *
 9: *********************************************************************/
10: #include     <stdio.h>
11:
12: /*********************************************************************
13: *
14: *    構造体 :     _TIME
15: *    機  能 :     時間データ構造の定義
16: *
17: *********************************************************************/
18: typedef struct _TIME {
19:     int hr;
20:     int min;
21:     int sec;
22:     int subs;
23: } TIME;
24:
25: void main (void);
26: void get_time (TIME *);
27: void time_msg (TIME *);
28: /*********************************************************************
29: *
30: *    関数名 :    main ()
31: *    機  能 :    時間表示機能の読み出し
32: *    入  力 :    なし
33: *    出  力 :    なし
34: *
35: *********************************************************************/
36: void main ()
37: {
38:     TIME time;
39:
40:     printf ("¥n *** 時間表示プログラム Ver.1.1 ***¥n¥n");
41:     printf ("準備ができたら 'G' キーを押して下さい.¥n");
42:     printf ("時間の表示をやめるには任意のキーを押して下さい.¥n");
43:     time.hr = 0;
44:     time.min = 0;
45:     time.sec = 0;
46:     time.subs = 0;
47:     get_time (&time);
48:     exit (0);
49: }
50:
51: /*********************************************************************
52: *
53: *    関数名 :    time_msg (ptr)
54: *    機  能 :    読み出した時間の表示
55: *    入  力 :    ptr ・・・・ 時間データの構造体へのポインタ
56: *    出  力 :    なし
57: *
58: *********************************************************************/
59: void time_msg (ptr)
60: TIME *ptr;
61: {
62:     printf ("time = %02d:%02d:%02d.%02d¥n", ptr -> hr
63:                                           , ptr -> min
64:                                           , ptr -> sec
65:                                           , ptr -> subs);
66: }
```

ます．

◆ **生成方法**

プログラム.exeは，次の手順で分割アセンブル/コンパイルして作成します．

masm /ML keyinsub;

cl －J －As keyin.c keyinsub

〔リスト7-14〕プログラム keyinsub.asm

```
 1: ;*****************************************************************
 2: ;
 3: ;       機    能 :  キー入力と時間の取り込み
 4: ;       ファンクション :  06H（コンソール直接入出力）
 5: ;                    0BH（キーボードのステータス・チェック）
 6: ;                    0CH（タイプ・アヘッド・バッファのクリア）
 7: ;                    2CH（時刻の読み出し）
 8: ;       生    成 :  masm /ML keyinsub;
 9: ;
10: ;*****************************************************************
11:             .MODEL  SMALL, C
12:             .CODE
13:             EXTRN   time_msg:NEAR
14: ;*****************************************************************
15: ;
16: ;       構造体 :  _TIME
17: ;       機    能 :  時間データ構造の定義
18: ;
19: ;*****************************************************************/
20: _TIME       STRUC
21: hr          DB      2 dup (?)
22: min         DB      2 dup (?)
23: sec         DB      2 dup (?)
24: subs        DB      2 dup (?)
25: _TIME       ENDS
26:
27: ;*****************************************************************
28: ;
29: ;       ルーチン名 :  get_time
30: ;       機    能 :  キーの入力により時刻データを読む
31: ;       func    :  06H（コンソールの直接入出力）
32: ;                  0BH（キーボードのステータス・チェック）
33: ;                  0CH（タイプ・アヘッド・バッファのクリア）
34: ;                  2CH（時刻の読み出し）
35: ;       入    力 :  arg1    時間データ構造体へのポインタ
36: ;       出    力 :  なし
37: ;
38: ;*****************************************************************
39: get_time    PROC    arg1:PTR
40:             mov     si, arg1                    ;構造体へのポインタ
41:             mov     ah, 06h                     ;ファンクション06H
42:             mov     dl, 0ffh                    ;入力モード
43: get_key:
44:             int     21h                         ;1文字入力
45:             cmp     al, 'g'                     ;'g'キー？
46:             je      next_read
47:             cmp     al, 'G'
48:             jne     get_key
49: next_read:
50:             mov     ah, 2Ch                     ;ファンクション2CH
51:             int     21h                         ;時刻の読み出し
52:
53:             mov     [si.hr],ch                  ;時
54:             mov     [si.min] ,cl                ;分
55:             cmp     [si.sec], dh
56:             je      next_read                   ;秒は同じか？
57:             mov     [si.sec] ,dh                ;秒
58:             mov     [si.subs] ,dl               ;1/100秒
59:             push    si
60:             call    time_msg                    ;時間データの表示
61:             pop     si
62:             mov     ah,0bh                      ;ファンクション0BH
63:             int     21h                         ;ステータス読み出し
64:             or      al,al                       ;何かのキーが押されたか
65:             je      next_read
66:             mov     ah, 0Ch                     ;ファンクション0CH
67:             mov     al, 0FFh
68:             int     21h                         ;タイプ・アヘッド・バッファを空にする
69:             ret
70: get_time    ENDP
71:             END
```

◆ 実行サンプル

リスト7-15 はプログラムkeyin.exeの実行例を示しています.

① 時間表示プログラムkeyin.exeを起動する.

② "g" キーの入力により時間データの表示を開始する.

③ 秒データに変化のあるときだけ時間データの表示が行われる.

④ その後に何かのキー入力があればプログラムkeyin.exeを終了する.

〔リスト7-15〕
プログラムkeyin.exeの実行例

```
R>keyin ⏎… プログラムの起動①

 *** 時間表示プログラム Ver.1.1 ***

準備ができたら 'G' キーを押して下さい.
時間の表示をやめるには任意のキーを押して下さい.
time = 18:32:58.00 …'g'キーの入力により時間データの表示を開始②
time = 18:32:59.00
time = 18:33:00.00
time = 18:33:01.00
time = 18:33:02.00
time = 18:33:03.00   秒データに変化があれば次々に時間データの表示を行う③
time = 18:33:04.00
time = 18:33:05.00
time = 18:33:06.00
time = 18:33:07.00
time = 18:33:08.00
time = 18:33:09.00 ← 任意のキーを押すとプログラムを終了④
R>
```

● BATファイルのネスト ●

MS-DOS(command.com)のバッチ処理では，単にBATファイルを入力デバイスにリダイレクト(置き換え)しているだけなので，BATファイルのネストはできません.

しかし幸いなことに，command.comも一つのコマンドであり，その起動オプションに/cを指定することによって，BATファイルのネスト処理が可能になります.

リストDは，バッチ処理のネストの例を示しています. 同リストでは，bat1.batの中からbat2.batを実行し，bat2.batが終了してから，再びbat1.batに戻って処理が続いていることが確認できます.

〔リストD〕

```
R>type bat1.bat ⏎ ……………… bat1.batの内容を確認
rem BAT1です.
command/c bat2.bat ……………… バッチのネスト部分
rem 戻りました.

R>type bat2.bat ⏎ ……………… bat2.batの内容を確認
rem BAT2です.

R>bat1 ⏎ ………………………… bat1.batの実行

R>rem BAT1です.

R>command/c bat2.bat

R>rem BAT2です. ……………… bat2が実行された

R>

R>rem 戻りました. ……… bat1が実行された

R>
```

日付と時刻を得る

リスト7-16(tm.asm)は，COPYキーの機能を横取りし，COPYキーが押されるたびに，その時点における日付と時刻の表示を行うプログラムtm.comのソース・リストです．

NECのPC-9801シリーズでは，COPYキーが押されるたびにINT 05Hのベクタが指しているアドレスに制御が移るので，このベクタを書き換えて日付と時刻の表示を行うプロシージャを起動します．

● tm.asm

同リストにおいて，プロシージャtime_setはプログラム・タイトルを表示したのち，ファンクション25Hを用いて割り込み処理ルーチン(int_05)のエントリ・アドレスをINT 05Hのベクタ・テーブルに登録します．

このあと，プログラムtm.comの最終セグメント・アドレスを計算して，そのアドレスをDXレジスタに設定して，ファンクション31Hによってtm.comをメモリに残したまま常駐終了します．

そのあと，COPYキーが押されるとINT 05Hの割り込みが発生し，プロシージャint_05に制御が移ります．このルーチンでは，まずファンクション2AHを用いて日付の読み出しを行い，各レジスタに返された年月日の値を，サブルーチンax_dispを用いて10進文字列に変換してメモリの該当する位置に格納します．格納された文字列は，サブルーチンstr outを用いてINT 29Hの内部割り込みにより表示を行います．

ここで，文字列の出力にはファンクション02Hやファンクション09Hが用意されていますが，このINT 05Hの割り込みルーチン内では，これらのファンクションが利用できないので注意が必要です．

日付の表示が終わったら，次にファンクション2CHを用いて時刻の読み出しを行い，日付と同様に10進文字列への変換を行ったのち，サブルーチンstr_outを用いてその時刻を表示します．

この割り込みルーチンでは，すべてのレジスタは保存しなければなりません．また，ここで使用したINT 05Hは公開されていない割り込みでもあり，ここで紹介した時間表示プログラムtm.comは，あまり「行儀の良い」プログラムではありません．したがって，すべてのPC-9801シリーズやMS-DOSの今後のバージョンでも問題なく動作するという保証はありません．

◆ 生成方法

プログラムtm.comは，次の手順でアセンブル/リンクして作成します．

```
masm /ML tm;
link /NOI tm;
exe2bin tm tm.com
```

このプログラムtm.comは，メモリに常駐させるためCOMモデルとして作成します(TSRプログラム)．COMモデルに限定してプログラムを記述すると，プログラム(特にプログラム最終アドレスの計算)の記述を簡潔に行うことができます．

〔リスト7-16〕プログラム tm.asm ①

```
 1: ;*****************************************************************
 2: ;
 3: ;    機    能 :  COPYキーの機能を横取りし時間データを表示する
 4: ;    ファンクション :  25H(割り込みベクタの設定)
 5: ;               2AH(日付の読み出し)
 6: ;               2CH(時刻の読み出し)
 7: ;               31H(プロセスの常駐終了)
 8: ;    生    成 :  masm /ML tm;
 9: ;               link /NOI tm;
10: ;               exe2bin tm tm.com
11: ;
12: ;*****************************************************************
13: CODE      SEGMENT
14:           ASSUME  CS:CODE, DS:CODE
15: ;*****************************************************************
16: ;
17: ;   ルーチン名 :  time_set
18: ;   機    能 :  時間表示ルーチンのベクタ設定
19: ;   func     :  25H(割り込みベクタの設定)
20: ;              31H(プロセスの常駐終了)
21: ;    入   力 :  なし
22: ;    出   力 :  AL ･･･ 終了コード00H(エラーなし常駐終了)
23: ;
24: ;*****************************************************************
```

〔リスト7-16〕プログラム tm.asm ②

```
25:              ORG     100h
26:
27: time_set     PROC
28:              push    cs
29:              pop     ds
30:              lea     dx, open_msg
31:              mov     ah, 09h
32:              int     21h                    ;プログラム・タイトル表示
33:              lea     dx, int_05
34:              mov     ah, 25h
35:              mov     al, 05h
36:              int     21h                    ;ベクタ・セット
37:              lea     dx, tail
38:              mov     cl, 4
39:              shr     dx, cl                 ;常駐パラグラフ
40:              inc     dx
41:              mov     ah, 31h
42:              xor     al, al
43:              int     21h                    ;常駐終了
44: time_set     ENDP
45:
46: ;**************************************************************************
47: ;
48: ;     ルーチン名:  int_05
49: ;     機    能:  COPYキー入力(INT 05H)で日付と時間を表示
50: ;     func    :  2AH(日付の読み出し)
51: ;                2CH(時刻の読み出し)
52: ;     入    力:  なし
53: ;     出    力:  なし
54: ;
55: ;**************************************************************************
56: int_05       PROC    FAR
57:              push    ax
58:              push    bx
59:              push    cx
60:              push    dx
61:              push    si
62:              push    ds
63:
64:              push    cs
65:              pop     ds
66:              lea     si, cr_data
67:              call    str_out
68:              lea     si, date_buff
69:              mov     ah, 2Ah                ;ファンクション 2AH
70:              int     21h                    ;日付の読み出し
71:              sub     cx, 1900
72:              mov     bl, cl
73:              call    ax_disp                ;16進数->10進文字
74:              mov     [si + 08h], bx         ;年 セット
75:              mov     bl, dh
76:              call    ax_disp                ;16進数->10進文字
77:              mov     [si + 0Bh], bx         ;月 セット
78:              mov     bl, dl
79:              call    ax_disp                ;16進数->10進文字
80:              mov     [si + 0Eh], bx         ;日 セット
81:              call    str_out
82:              lea     si, time_buff
83:              mov     ah, 2Ch                ;ファンクション 2CH
84:              int     21h                    ;時刻の読み出し
85:              mov     bl, ch
86:              call    ax_disp                ;16進数->10進文字
87:              mov     [si + 6], bx           ;時 セット
88:              mov     bl, cl
89:              call    ax_disp                ;16進数->10進文字
90:              mov     [si + 9], bx           ;分 セット
91:              mov     bl, dh
92:              call    ax_disp                ;16進数->10進文字
93:              mov     [si + 0Ch], bx         ;秒 セット
94:              mov     bl, dl
95:              call    ax_disp                ;16進数->10進文字
96:              mov     [si + 0Fh], bx         ;1/100秒 セット
97:              call    str_out
98:              pop     ds
99:              pop     si
```

〔リスト7-16〕プログラム tm.asm ③

```
100:                pop     dx
101:                pop     cx
102:                pop     bx
103:                pop     ax
104:                iret
105: int_05         ENDP
106:
107: ;*************************************************************************
108: ;
109: ;    ルーチン名 :  ax_disp
110: ;    機    能 :  16進数を10進数の文字に変換
111: ;    入    力 :  BL ･･･ 16進数(1バイト)
112: ;    出    力 :  BX ･･･ 10進文字列
113: ;
114: ;*************************************************************************
115: ax_disp        PROC
116:                push    ax
117:                xor     bh, bh
118:                mov     ax, bx
119:                mov     bh, 10
120:                div     bh
121:                add     ax, 3030h
122:                mov     bx, ax
123:                pop     ax
124:                ret
125: ax_disp        ENDP
126:
127: ;*************************************************************************
128: ;
129: ;    ルーチン名 :  str_out
130: ;    機    能 :  文字列の表示
131: ;    入    力 :  SI ･･･ 文字列へのポインタ
132: ;    出    力 :  なし
133: ;
134: ;*************************************************************************
135: str_out        PROC
136:                push    ax
137:                push    si
138: str_loop:
139:                mov     al, [si]
140:                cmp     al, '$'
141:                je      exit
142:                int     29h
143:                inc     si
144:                jmp     str_loop
145: exit:
146:                pop     si
147:                pop     ax
148:                ret
149: str_out        ENDP
150:
151: ;*************************************************************************
152: ;
153: ;    データ名 :  open_msg
154: ;    機    能 :  プログラム・タイトル
155: ;
156: ;    データ名 :  cr_data
157: ;    機    能 :  1行改行
158: ;
159: ;    データ名 :  date_buff
160: ;    機    能 :  日付データ・バッファ
161: ;
162: ;    データ名 :  time_buff
163: ;    機    能 :  時間データ・バッファ
164: ;
165: ;*************************************************************************
166: open_msg       DB      0Dh, 0Ah, '時間表示プログラムが常駐しました'
167:                DB      0Dh, 0Ah, '時間を表示する際にはCOPYキーを押して下さい'
168:                DB      0Dh, 0Ah, '$'
169: cr_data        DB      0Dh, 0Ah, '$'
170: date_buff      DB      'date= 1989-01-01',0Dh,0Ah,'$'
171: time_buff      DB      'time= 00:00:00.00',0Dh,0Ah,'$'
172: tail           LABEL   NEAR
173: CODE           ENDS
174:                END     time_set
```

◆ **実行サンプル**

リスト7-17は時間表示プログラムtm.comの実行例を示しています．

① まず，プログラムtm.comを起動して時間表示ルーチンをメモリに常駐させる．

② 時間を知りたいときにCOPYキーを押すと，随時時間データを表示させることができる．

DOSのバージョンを取り出す

リスト7-18(gver.c)および**リスト7-19**(gversub.asm)は，DOSのバージョン番号を読み出して表示するプログラムgver.exeのソース・リストです．

● **gver.c**

リスト7-18はプログラムgver.exeのCソース部分です．同リストにおいて，構造体_VERはファンクション30Hにおいて読み出されたバージョン番号を格納するデータ領域の構造を定義しています．

関数mainでは，最初にプログラム・タイトルを表示したのち，コマンド・ライン・パラメータの解析を行います．パラメータの解析では，まずコマンド・ライ

〔リスト7-17〕
プログラムtm.comの実行例

```
R>tm ⏎… プログラムを起動しメモリに常駐させる①
R>
date= 1988-12-10 ┐
time= 12:03:31.00 ┘ COPY キーの入力によって時間が表示される…②
R>
```

〔リスト7-18〕**プログラムgver.c ①**

```
 1: /**********************************************************************
 2: *
 3: *    機    能:  バージョン番号の読み出しと表示
 4: *    サ    ブ:  gversub.asm
 5: *    生    成:  masm /ML gversub;
 6: *              cl -J -AS gver.c gversub
 7: *    使用方法:  gver
 8: *
 9: **********************************************************************/
10: #include     <stdio.h>
11:
12: /**********************************************************************
13: *
14: *    構造体:  _VER
15: *    機  能:  バージョン番号データの構造定義
16: *
17: **********************************************************************/
18: typedef struct _VER {
19:     unsigned int ver1;        /* 整数部 */
20:     unsigned int ver2;        /* 小数部 */
21:     unsigned int oem;         /* OEM番号 */
22:     unsigned long ser;        /* シリアル番号 */
23: } VER;
24:
25: void main (int, char **);
26: void ver_read (void);
27: void ver_msg (VER *);
28: int  func_30 (VER *);           /* バージョン番号の読み出し */
29: /**********************************************************************
30: *
31: *    関数名:  main (argc, argv)
32: *    機  能:  コマンドライン・パラメータの解析
33: *    機  能:  バージョン番号の読み出しと表示
34: *    入  力:  int argc      ・・・・  コマンドライン・パラメータの数
35: *              char *argv[]  ・・・・  パラメータ文字列へのポインタ
36: *    出  力:  なし
37: *
38: **********************************************************************/
39: void main (argc, argv)
40: int argc;
41: char **argv;
42: {
```

ンのパラメータの個数を調べ，もし不要なパラメータがあればその旨を表示します．そして，関数 ver_read を用いてバージョン番号の読み出しと表示を行います．

関数 ver_read では，構造体の確保を行ったのち，関数(サブルーチン)func_30 にその構造体へのポインタを渡してバージョン番号の読み出しを行います．読み出されたバージョン番号は構造体に格納されて返されるので，関数 ver_msg に対して，その構造体へのポインタを渡してバージョン番号の表示を行っています．

関数 ver_msg では，ポインタで渡された構造体を参照し，その中の各メンバに格納されている DOS のバージョン番号や OEM のシリアル番号などを表示します．

● gversub.asm

リスト7-19 はプログラム gver.exe のアセンブリ・ソース部分です．同リストにおいて，ストラクチャ _VER は，**リスト7-18** の C ソース内の構造体 _VER に対応していて，バージョン番号を格納するデータ領域の構造を定義しています．

サブルーチン(関数)func_30 は，ファンクション 30H を用いて DOS のバージョン番号を読み出し，ポインタ argl で指定された構造体の中の各メンバにバージョン番号の整数部，小数部，OEM 番号，DOS のシリアル番号をそれぞれ格納して戻ります．

◆ **生成方法**

プログラム gver.exe は，次の手順で分割アセンブル/コンパイルして作成します．

masm /ML gversub;

cl －J －As gver.c gversub

◆ **実行サンプル**

リスト7-20 はプログラム gver.exe の実行例を示しています．この中で，DOS のシリアル番号はどの DOS バージョンでも "0" が表示され，現在のところサポートされていないようです．

〔リスト7-18〕プログラム gver.c ②

```
43:
44:     printf ("¥n *** バージョン番号表示プログラム Ver.1.1 ***¥n¥n");
45:     if (argc != 1) {
46:         printf ("パラメータは不要です.¥n");
47:     }
48:     ver_read ();
49:     printf ("¥n");
50:     exit (0);
51: }
52:
53: /*********************************************************************
54: *
55: *    関数名:  ver_read ()
56: *    機  能:  バージョン番号の読み出しと表示
57: *    入  力:  なし
58: *    出  力:  なし
59: *
60: *********************************************************************/
61: void ver_read ()
62: {
63:     VER data;
64:
65:     func_30 (&data);
66:     ver_msg (&data);
67: }
68:
69: /*********************************************************************
70: *
71: *    関数名:  ver_msg (ptr)
72: *    機  能:  バージョン番号の表示
73: *    入  力:  ptr ･･･ 構造体へのポインタ
74: *    出  力:  なし
75: *
76: *********************************************************************/
77: void ver_msg (ptr)
78: VER *ptr;
79: {
80:     printf ("DOSのバージョン番号      %d.%d¥n", ptr -> ver1, ptr -> ver2);
81:     printf ("OEMのシリアル番号        %d¥n", ptr -> oem);
82:     printf ("DOSのシリアル番号        %ld¥n", ptr -> ser);
83: }
```

〔リスト7-19〕プログラム gversub.asm

```
 1: ;*************************************************************************
 2: ;
 3: ;      機　　能：  バージョン番号取得サブルーチン
 4: ;      ファンクション：  30H（バージョン番号の取得）
 5: ;      生　　成：  masm /ML gversub;
 6: ;
 7: ;*************************************************************************
 8:                 .MODEL   SMALL, C
 9:                 .CODE
10: ;*************************************************************************
11: ;
12: ;      構造体：  _VER
13: ;      機　　能：  バージョン番号情報構造の定義
14: ;
15: ;*************************************************************************
16: _VER            STRUC
17: ver1            DW       ?
18: ver2            DW       ?
19: oem             DW       ?
20: ser1            DW       ?
21: ser2            DW       ?
22: _VER            ENDS
23:
24: ;*************************************************************************
25:
26: ;      ルーチン名：  func_30
27: ;      機　　能：  バージョン番号を返す
28: ;      func　 ：  30H（バージョン番号の取得）
29: ;      入　　力：  arg1 ‥‥バージョン番号構造体へのポインタ
30: ;      出　　力：  なし
31: ;
32: ;*************************************************************************
33: func_30         PROC     arg1:PTR
34:                 push     si
35:                 mov      si, arg1
36:                 mov      ah, 30h                    ;ファンクション 30H
37:                 int      21h
38:                 push     ax
39:                 xor      ah, ah
40:                 mov      [si.ver1], ax              ;整数部
41:                 pop      ax
42:                 mov      al, ah
43:                 xor      ah, ah
44:                 mov      [si.ver2], ax              ;小数部
45:                 push     bx
46:                 mov      bl, bh
47:                 xor      bh, bh
48:                 mov      [si.oem], bx               ;OEM番号
49:                 pop      bx
50:                 xor      bh, bh
51:                 mov      [si.ser1], bx              ;上位ワード
52:                 mov      [si.ser2], cx              ;下位ワード
53:                 pop      si
54:                 ret
55: func_30         ENDP
56:                 END
```

〔リスト7-20〕
プログラム gver.exe の実行例

```
R>gver ⏎…プログラムの起動

  *** バージョン番号表示プログラム Ver.1.1 ***

DOSのバージョン番号         3.30
OEMのシリアル番号       ‥‥  255
DOSのシリアル番号       ‥‥  0

R>
```

7-3 メモリ/プロセスの操作

子プロセスを実行する

リスト7-21(child.asm)は，ファンクション4B00Hを用いて子プロセスの実行を行い，子プロセスのプロセス終了コードを表示するプログラムchild.comのソース・リストです．

● child.asm

同リストにおいて，プロシージャchild_loadは，最初にプログラム・タイトルを表示したのち，子プロセスのファイル名やコマンド・ラインのコピーを行い，そのあとにファンクション4AHを用いて子プロセスをロードするためのメモリ・ブロックの確保を行います．

次に，パラメータ・ブロックの設定を行ってからファンクション4B00Hを用いて子プロセスのロードと実行を行います．ここで，ファイル名が違っているなどファンクション4B00Hでエラーが発生した場合には，エラー・メッセージを表示してプログラムchild.asmをエラー・ストップします．子プロセスが正常に実行され終了したら，ファンクション4DHによって子プロセスの終了コードを調べその状態を表示します．

サブルーチン(プロシージャ)put_axは，AXレジスタに渡された内容を4桁の16進数で表示します．同様に，サブルーチンput_alはALレジスタの内容を2桁の16進数で表示し，サブルーチンput_1は，ALレジスタの下位4ビットの内容を1桁の16進数として表示します．

◆ 生成方法

プログラムchild.comは，次の手順でアセンブルして作成します．

```
masm /ML child;
link /NOI child;
exe2bin child child.com
```

このプログラムchild.comは，プログラムが小さく64Kバイト以下に収まるため，ユーティリティexe2binを用いてCOMモデルに変換します．

COMモデルでは，ESやDSなどのセグメント・レジスタがPSPやコード・セグメントを指しているため，プログラムを簡潔に記述することができます．

〔リスト7-21〕
プログラム
child.asm ①

```
 1: ;*****************************************************************
 2: ;
 3: ;     機    能 :    子プロセスのロードと実行
 4: ;     ファンクション :    02H  (文字の出力)
 5: ;                    09H  (文字列の出力)
 6: ;                    4AH  (メモリ・ブロックのサイズ変更)
 7: ;                    4B00H(プロセスのロードと実行)
 8: ;                    4CH  (プロセス終了)
 9: ;     生    成       masm /ML child;
10: ;                    link /NOI child;
11: ;                    exe2bin child child.com
12: ;
13: ;*****************************************************************
14: CODE        SEGMENT
15:             ASSUME  CS:CODE, DS:CODE
16: ;*****************************************************************
17: ;
18: ;      ルーチン名 :   child_load
19: ;      機    能 :   メモリブロックの確保と子プロセスの実行
20: ;      func    :   09H  (文字列の出力)
21: ;                  4AH  (メモリ・ブロックのサイズ変更)
22: ;                  4B00H プロセスのロードと実行)
23: ;                  4CH  (プロセス終了)
24: ;      入    力:   コマンド・ライン(子プロセス名および引数)
25: ;      出    力:   なし
26: ;
27: ;*****************************************************************
28:             ORG     100h
29:
30: child_load  PROC
31:             lea     sp, stack_bot
32:             lea     dx, open_msg
33:             mov     ah, 09h                    ;func 09H
34:             int     21h                        ;プログラム・タイトル
35:             mov     si, 82h
36:             mov     cl [si-2]
37:             xor     ch, ch
38:             or      cl, cl
```

〔リスト7-21〕
プログラム
child.asm ②

```
 39:            je       error                    ;ファイル名有り?
 40:            lea      di, file_name
 41:            xor      dl, dl
 42: next_char:
 43:            movsb
 44:            inc      dl
 45:            cmp      BYTE PTR [di - 1], ' '   ;空白
 46:            je       char_end
 47:            loop     next_char
 48:
 49: char_end:
 50:            mov      BYTE PTR [di - 1], 00h   ;asciz
 51             push     cx
 52:            mov      di, offset 82h
 53:            mov      cl, [di - 2]
 54:            sub      cl, dl
 55:            mov      [di - 2], cl             ;文字数セット
 56:            pop      cx
 57:            rep      movsb
 58:            mov      byte ptr [di], 0dh       ;パラメータセット
 59:
 60:            lea      bx, tail
 61:            mov      cl, 4
 62:            shr      bx, cl                   ;パラグラフ
 63:            inc      bx
 64:            mov      ah, 4Ah                  ;func 4AH
 65:            int      21h                      ;メモリブロック変更
 66:            jc       error
 67:
 68:            mov      bx, ds
 69:            mov      para[4], bx
 70:            mov      para[8], bx
 71:            mov      para[12], bx             ;パラメータブロック
 72:            mov      dx, offset file_name
 73:            mov      bx, offset para
 74:            mov      ax, 4B00h                ;ロード&実行
 75:            int      21h                      ;exec コール
 76:            jnc      no_error
 77: error:
 78:            lea      dx, err_msg
 79:            mov      ah, 09h                  ;func 09H
 80:            int      21h
 81:            mov      al, 02h                  ;リターンコード
 82:            jmp      fini
 83:
 84: no_error:
 85:            lea      dx, end_msg
 86:            mov      ah, 09h                  ;func 09H
 87:            int      21h                      ;終了メッセージ
 88:            mov      ah, 4Dh
 89:            int      21h                      ;Get 終了コード
 90:            lea      dx, norm_msg
 91:            cmp      ah, 0
 92:            je       exit                     ;正常終了
 93:            lea      dx, ctrl_msg
 94:            cmp      ah, 1
 95:            je       exit                     ;ctrl-C
 96:            lea      dx, fatal_msg
 97:            cmp      ah, 2                    ;致命的エラー
 98:            je       exit
 99:            lea      dx, keep_msg
100:            cmp      ah, 3                    ;常駐終了
101:            je       exit
102:            lea      dx, etc_msg
103: exit:
104:            push     ax
105:            mov      ah, 09h
106:            int      21h                      ;終了ステータス
107:            lea      dx, rtc_msg
108:            int      21h                      ;'終了コード='
109:            pop      ax
110:            call     put_ax                   ;終了コード表示
111:            mov      al, 00                   ;リターンコード
112: fini:
113:            push     ax
114:            lea      dx, cr_msg
115:            mov      ah, 09h
116:            int      21h
```

〔リスト7-21〕
プログラム
child.asm ③

```
117:                    pop     ax
118:                    mov     ah, 4Ch                 ;ファンクション4CH
119:                    int     21h                     ;プログラムの正常終了
120: child_load         ENDP
121: ;
122: ;**********************************************************************
123: ;
124: ;    ルーチン名 :   put_ax
125: ;    機   能 :   AXの内容を表示
126: ;    入   力 :   AX ‥ 16進数 (2バイト)
127: ;    出   力 :   なし
128: ;
129: ;**********************************************************************
130: put_ax             PROC
131:                    push    ax
132:                    push    cx
133:
134:                    push    ax
135:                    mov     al, ah
136:                    call    put_al          ;上位バイト
137:                    pop     ax
138:                    call    put_al                  ;下位バイト
139:
140:                    pop     cx
141:                    pop     ax
142:                    ret
143: put_ax             ENDP
144:
145: ;**********************************************************************
146: ;
147: ;    ルーチン名 :   put_al
148: ;    機   能 :   ALレジスタの表示
149: ;    入   力 :   AL ‥ 16進数 (1バイト)
150: ;    出   力 :   なし
151: ;
152: ;**********************************************************************
153: put_al             PROC
154:                    push    ax
155:                    push    cx
156:
157:                    push    ax
158:                    mov     cl, 4
159:                    shr     al, cl
160:                    call    put1
161:                    pop     ax
162:                    call    put1
163:
164:                    pop     cx
165:                    pop     ax
166:                    ret
167: put_al             ENDP
168:
169: ;**********************************************************************
170: ;
171: ;    ルーチン名 :   put_1
172: ;    機   能 :   1桁16進数を表示
173: ;    func     :   02H (文字の出力)
174: ;    入   力 :   AL ‥ 16進数 (1桁)
175: ;    出   力 :   なし
176: ;
177: ;**********************************************************************
178: put1               PROC
179:                    push    ax
180:                    push    dx
181:                    and     al, 0Fh
182:                    cmp     al, 0Ah
183:                    jb      num
184:                    add     al, 07h
185: num:
186:                    add     al, '0'
187:                    mov     dl, al
188:                    mov     ah, 02h
189:                    int     21h
190:                    pop     dx
191:                    pop     ax
192:                    ret
193: put1               ENDP
194:
```

〔リスト7-21〕
プログラム
child.asm ④

```
195: ;***********************************************************************
196: ;
197: ;      データ名 :  open_msg
198: ;      機   能 :  プログラム・タイトル
199: ;
200: ;      データ名 :  err_msg
201: ;      機   能 :  エラー・メッセージ
202: ;
203: ;      データ名 :  end_msg
204: ;      機   能 :  終了メッセージ
205: ;
206: ;      データ名 :  norm_msg
207: ;      機   能 :  正常終了メッセージ
208: ;
209: ;      データ名 :  ctrl_msg
210: ;      機   能 :  ctrl-C中断メッセージ
211: ;
212: ;      データ名 :  fatal_msg
213: ;      機   能 :  致命的エラー中断メッセージ
214: ;
215: ;      データ名 :  keep_msg
216: ;      機   能 :  常駐終了メッセージ
217: ;
218: ;      データ名 :  ect_msg
219: ;      機   能 :  その他の状態終了メッセージ
220: ;
221: ;      データ名 :  rtc_msg
222: ;      機   能 :  終了コード・メッセージ
223: ;
224: ;      データ名 :  cr_msg
225: ;      機   能 :  CRコード
226: ;
227: ;      データ名 :  para
228: ;      機   能 :  パラメータ・ブロック
229: ;
230: ;      データ名 :  file_name
231: ;      機   能 :  ファイル名バッファ
232: ;
233: ;      データ名 :  stack_bot
234: ;      機   能 :  スタック・ボトム
235: ;
236: ;***********************************************************************
237: open_msg     DB        0Dh, 0Ah
238:              DB        'プロセス終了コード表示プログラム Ver.1.1'
239:              DB        0Dh, 0Ah, '$'
240: err_msg      DB        '子プロセスをロードできません.', 0Dh, 0Ah, '$'
241: end_msg      DB        0Dh, 0Ah, '子プロセスの実行を終了しました.'
242:              DB        0Dh, 0Ah, '子プロセスは', '$'
243: norm_msg     DB        '正常終了です.', '$'
244: ctrl_msg     DB        'ctrl-C入力による中断です.', '$'
245: fatal_msg    DB        '致命的エラーによる中断です.', '$'
246: keep_msg     DB        '常駐終了です.', '$'
247: etc_msg      DB        '未定義の状態で終了しました.', '$'
248: rtc_msg      DB        0Dh, 0Ah, '終了コード =', '$'
249: cr_msg       DB        0Dh, 0Ah, '$'
250: para         DW        00h, 80h, ?, 5ch, ?, 6ch, ?
251: file_name    DB        64 dup (?)
252:              DW        80h dup (?)
253: stack_bot    DW        ?
254: tail         LABEL     NEAR
255: CODE         ENDS
256:              END       child_load
```

◆ 実行サンプル

リスト7-22はプログラムchild.comの実行例を示しています.

① プログラムchild.comの子プロセスとしてマクロ・アセンブラmasm.exeを起動する. ここでは, masm.exeに対してアセンブルすべきソース・ファイル名を指定していない.

② そこで, masm.exeはソース・ファイル名の入力を促してくるのでchild.asmを指定する.

③ その他のmasm.exeからのメッセージも通常どおりに表示される.

④ 子プロセスであるmasm.exeの実行が終わると, child.comからのメッセージが表示される.

⑤ ここで, 子プロセスであるmasm.exeの終了コードが表示され, 子プロセスがどのような状態で終了したかを知ることができる.

⑥ 次にchild.comの子プロセスとしてmasm.exeにパラメータ(ファイル名)を与えて起動する.

〔リスト7-22〕プログラム child.com の実行例

```
R>child masm.exe ⏎… 子プロセスとして masm.exe を指定して起動①

プロセス終了コード表示プログラム Ver.1.1
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.  ← masm.exe から出力されるメッセージ③

Source filename [.ASM]: child; ⏎… ファイル名の入力要求②

  49342 + 147872 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors

子プロセスの実行を終了しました.  ← child.com から表示されるメッセージ④
子プロセスは正常終了です.
終了コード  = 0000 … 終了コードの表示⑤

R>child masm.exe child; ⏎… masm.exe にパラメータを与えて起動⑥

プロセス終了コード表示プログラム Ver.1.1
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.  ← masm.exe からのメッセージ⑦

  49342 + 147872 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors

子プロセスの実行を終了しました.
子プロセスは正常終了です.
終了コード  = 0000 … 正常終了した⑧

R>child masm.exe child; ⏎… 再び masm.exe にパラメータを与えて起動⑨

プロセス終了コード表示プログラム Ver.1.1
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.

^C … 処理の途中で ctrl-C を入力する⑩

  49342 + 147872 Bytes symbol space free

子プロセスの実行を終了しました.
子プロセスは正常終了です.
終了コード  = 000B … 終了コードが異なる⑪

R>child dump.com child.com ⏎… 子プロセスとして dump.com にパラメータを与えて起動⑫

プロセス終了コード表示プログラム Ver.1.1

Dump Version 2.1

00000000  8D 26 4C 04 8D 16 FF 01-B4 09 CD 21 BE 82 00 8A   .&L.....ｴ.ﾍ!ｾ..貝
00000010  4C FE 32 ED 0A C9 74 52-8D 3E 0C 03 32 D2 A4 FE   .2..ﾉtR.>..2ﾒ..
00000020  C2 80 7D FF 20 74 02 E2-F5 C6 45 FF 00 51 BF 82   ﾂ.}. t.模ﾆE..Qｿ.
                          .
                          .
                          .
00000160  8F 49 97 B9 82 B5 82 DC-82 B5 82 BD 2E 0D 0A 8E   終了しました...子
00000170  71 83 76 83 8D 83 5A 83-58 82 CD 24 90 B3 8F ED    プロセスは$正常
00000180  8F 49 97 B9 82 C5 82 B7-^C … ctrl-C を入力⑬

子プロセスの実行を終了しました.  ← 子プロセス終了メッセージ⑭
子プロセスは ctrl-C 入力による中断です.
終了コード  = 0100

R>child divset.com ⏎… 子プロセスとして divset.com を起動する⑮

プロセス終了コード表示プログラム Ver.1.1

除算エラー処理ルーチンが常駐しました.

子プロセスの実行を終了しました.  ← divset.com は常駐終了した⑯
子プロセスは常駐終了です.
終了コード  = 0300
```

⑦ この場合も masm.exe から通常のメッセージが表示される.

⑧ 終了コードを見ると正常に終了したことが確認される.

⑨ 再び child.com の子プロセスとして masm.exe にパラメータを与えて起動する.

⑩ masm.exe の処理の途中で ctrl-C を入力して masm.exe を中断する.

⑪ すると, masm.exe からは正常終了のときと異なる終了コード(000BH)が返される. ここで masm.exe は, ctrl-C 入力の中断にもかかわらず AH レジスタには 00H を返していることも確認される.

⑫ 次に, child.com の子プロセスとして dump.com にパラメータ(ファイル名)を与えて起動する.

⑬ dump.com の処理中に ctrl-C を入力して dump.com を中断する.

⑭ すると, child.com の終了メッセージが「ctrl-C 中断」のメッセージに変わる. また, 終了コードも 0100H となって ctrl-C による中断であることを表している.

⑮ ここで, child.com の子プロセスとして, あとで紹介する divset.com を起動する. プログラム devset.com はメモリに常駐したまま終了するプログラムである.

⑯ 終了コードから, divset.com はメモリに常駐したまま終了したことが確認される.

■ メモリ・ブロックの操作を行う

リスト7-23(maloc.c)および**リスト7-24**(malocsub.asm)は, ファンクション 48H～4AH を用いてメモリ・ブロックの操作を行い, メモリ管理情報の表示を行うプログラム maloc.exe のソース・リストです.

● maloc.c

リスト7-23 はプログラム maloc.exe の C ソース部分です. 同リストにおいて, 関数 main では最初にプログラム・タイトルの表示を行い, つづいて関数 mem_disp によってメモリ・ブロックの操作とメモリ管理情報の表示を行います.

関数 mem_disp では, 関数(サブルーチン) func_62 を用いて, プログラム maloc.exe の PSP のセグメント・アドレスを得ます. PSP のセグメント・アドレスの直前のセグメントにはメモリ管理情報が存在するので, 関数 sub_eseg を用いてセグメント・アドレスを 16 バイト分だけ差し引き, そのセグメント・アドレスをもって関数 mdump によってメモリ管理情報をメモリ・ダンプします.

次に, 関数 func_4a を用いて PSP のベース・セグメントから 20000H バイト(セグメント 2000H)分のメモリ・ブロックのサイズ変更を行います. ここで, もし関数 func_4a でエラーが発生したら, エラー・メッセージを表示してプログラム maloc.exe をエラー・ストップします. メモリ ブロックのサイズ変更に成功したら, まえのメモリ管理情報の内容と新たに作成されたメモリ管理情報の内容を, 関数 mdump を用いてメモリ・ダンプします.

次に, 関数 func_48 を用いてメモリ・ブロックの割り当てを要求します. ここでは, 要求するメモリ・ブロックとして 2000H バイト(セグメント 200H)を要求しています. ここでも, 関数 func_48 でエラーが発生したら, エラー・メッセージを表示してプログラム maloc.exe をエラー・ストップします. もし, メモリ・ブロックの割り当てに成功したら関数 mdump を用いて, 関数 func_48 の実行前後におけるメモリ管理情報の表示を行います.

次に, 関数 func_48 を用いて割り当てられたメモリ・ブロックを関数 func_49 を用いて解放します. そして, やはり関数 mdump を用いて, 関数 func_49 の実行前後におけるメモリ管理情報の表示を行います.

関数 mdump は, FAR ポインタで渡されたメモリ内容のダンプ・リストを表示します. ここで, プログラム maloc.exe では 16 バイトごとのメモリ・ダンプしか必要ありませんが, この関数 mdump は, 汎用的な関数として使えるように 256 バイトごとのメモリ・ダンプにも対応しています.

関数 para_dump は, 16 バイトごとのメモリ・ダンプを行います. この関数 para_dump では, メモリ・アドレスを FAR ポインタとして受け取り, そのアドレスとメモリ内容の表示を行います.

● malocsub.asm

リスト7-24 はプログラム maloc.exe のアセンブリ・ソース部分です. 同リストにおいて, サブルーチン(関数) func_62 はファンクション 62H を用いてそのプログラムの PSP セグメントのアドレスを FAR ポインタとして DX : AX レジスタに返します.

サブルーチン func_48 は, ファンクション 48H を用いてメモリ・ブロックの割り当てを行います. このサブルーチンでは, PSP へのポインタ(arg1)と必要な割り当てサイズ(arg2 のパラグラフ数)を引数として受け取り, ファンクション 48H を実行します. 得られたメモリの先頭のセグメント・アドレスは FAR ポインタとして DX : AX レジスタに返します. ここで, もしファンクション 48H でエラーが発生した場合には DX : AX レジスタを "0" にして返します.

〔リスト7-23〕プログラム maloc.c ①

```
 1: /***********************************************************************
 2: *
 3: *   機     能 :   メモリブロックの操作と表示
 4: *   サ     ブ :   malocsub.asm
 5: *   生     成 :   masm /ML malocsub;
 6: *                 cl -J -AS maloc.c malocsub
 7: *   使用方法 :   maloc
 8: *
 9: ***********************************************************************/
10: #include      <stdio.h>
11: #include      <dos.h>
12:
13: void main (void);
14: void mem_disp (void);
15: void mdump (char far *, unsigned int);
16: void para_dump (char far *);
17: char far *func_62 (void);                   /* PSPアドレスの取得 */
18: char far *func_48 (char far *, int);   /* メモリ・ブロックの割り当て */
19: char far *func_49 (char far *);          /* メモリ・ブロックの解放 */
20: char far *func_4a (char far *, int);    /* メモリ・ブロックのサイズ変更 */
21: char far *add_eseg (char far *, int);
22: char far *sub_eseg (char far *, int);
23: /***********************************************************************
24: *
25: *    関数名 :   main ()
26: *    機   能 :   メモリ・ブロック表示関数の呼び出し
27: *    入   力 :   なし
28: *    出   力 :   なし
29: *
30: ***********************************************************************/
31: void main ()
32: {
33:
34:     printf (" *** メモリ・ブロック表示プログラム Ver.1.1 *** ¥n");
35:     mem_disp ();
36:     printf ("¥n");
37:     exit (0);
38: }
39:
40: /***********************************************************************
41: *
42: *    関数名 :   mem_disp ()
43: *    機   能 :   メモリ・ブロックの操作と表示
44: *    入   力 :   なし
45: *    出   力 :   なし
46: *
47: ***********************************************************************/
48: void mem_disp ()
49: {
50:     char far *ptr0;
51:     char far *ptr1;
52:     char far *ptr2;
53:     char far *eseg;
54:
55:     printf ("¥n起動時のメモリ・ブロックの内容¥n");
56:     eseg = func_62 ();
57:     ptr1 = sub_eseg (eseg, 1);
58:     mdump (ptr1, 16);
59:
60:     if ((eseg = func_4a (eseg, 0x2000)) == 0) {
61:         printf ("ファンクション 4AH でエラーが発生しました.¥n");
62:         exit (1);
63:     }
64:     printf ("¥nファンクション 4AH を実行しました.¥n");
65:     printf ("現時点のメモリ・ブロックの内容¥n");
66:     mdump (ptr1, 16);
67:     printf ("新たなメモリ・ブロックの内容¥n");
68:     ptr1 = add_eseg (eseg, 0x2000);
69:     mdump (ptr1, 16);
70:
71:     ptr0 = ptr1;
72:     if ((ptr1 = func_48 (ptr0, 0x200)) == 0) {
73:         printf ("ファンクション 48H でエラーが発生しました.¥n");
74:         exit (2);
75:     }
76:     printf ("¥nファンクション 48H を実行しました.¥n");
```

〔リスト7-23〕プログラム maloc.c ②

```
 77:    printf ("現時点のメモリ・ブロックの内容¥n");
 78:    mdump (ptr0, 16);
 79:    printf ("新たなメモリ・ブロックの内容¥n");
 80:    ptr2 = add_eseg (ptr1, 0x200);
 81:    mdump (ptr2, 16);
 82:
 83:    if ((ptr1 = func_49 (ptr0)) == 0) {
 84:        printf ("ファンクション 49H でエラーが発生しました.¥n");
 85:        exit (3);
 86:    }
 87:    printf ("¥nファンクション 49H を実行しました.¥n");
 88:    printf ("現時点のメモリ・ブロックの内容¥n");
 89:    mdump (ptr0, 16);
 90:    printf ("前のメモリ・ブロックの内容¥n");
 91:    mdump (ptr2, 16);
 92: }
 93:
 94: /*****************************************************************
 95: *
 96: *    関数名:   mdump (ptr)
 97: *    機  能:   メモリ・ダンプ (256バイト単位)
 98: *    入  力:   ptr      データ領域へのポインタ
 99: *    出  力:   なし
100: *
101: *****************************************************************/
102: void mdump (ptr, bytes)
103: char far *ptr;
104: unsigned int bytes;
105: {
106:    char far *i;
107:
108:    do {
109:        for (i = ptr; i < ptr + 0x100; i += 16) {
110:            para_dump (i);
111:            if ((bytes -= 16) <= 0) {
112:                return;
113:            }
114:        }
115:        ptr += 0x100;
116:        printf ("¥n");
117:    } while (bytes > 0);
118: }
119:
120: /*****************************************************************
121: *
122: *    関数名:   para_dump (ptr)
123: *    機  能:   メモリ・ダンプ (16バイト単位)
124: *    入  力:   ptr      データ領域へのポインタ
125: *    出  力:   なし
126: *
127: *****************************************************************/
128: void para_dump (ptr)
129: char far *ptr;
130: {
131:    char far *i;
132:    char c;
133:
134:    printf (" %04X:%04X ", FP_SEG(ptr), FP_OFF(ptr));
135:    for (i = ptr; i < ptr + 8; i++) {
136:        printf ("%02X ", *i & 0x00FF);
137:    }
138:    printf ("- ");
139:    for (;i < ptr + 16; i++) {
140:        printf ("%02X ", *i & 0x00FF);
141:    }
142:    printf ("  ");
143:    for (i = ptr; i < ptr + 16; i++) {
144:        c = *i;
145:        if (c < ' ' || c >= 0x7F) {
146:            c = '.';
147:        }
148:        printf ("%c", c);
149:    }
150:    printf ("¥n");
151: }
```

〔リスト7-24〕
プログラム
malocsub.asm ①

```
 1: ;*******************************************************************
 2: ;
 3: ;   機    能:   メモリ・ブロック操作サブルーチン
 4: ;   ファンクション   48H（メモリブロックの割り当て）
 5: ;                   49H（メモリ・ブロックの解放）
 6: ;                   4AH（メモリ・ブロックのサイズ変更）
 7: ;                   62H（ＰＳＰアドレスの取得）
 8: ;   生    成:   masm /ML malocsub;
 9: ;
10: ;*******************************************************************
11:             .MODEL  SMALL, C
12:             .CODE
13: ;*******************************************************************
14: ;
15: ;       ルーチン名:   get_eseg
16: ;       機   能:ESレジスタ内容の取得
17: ;       func   :62H（ＰＳＰアドレスの取得）
18: ;       入   力:なし
19: ;       出   力:DX   ESレジスタ内容
20: ;                 AX   0000H
21: ;
22: ;*******************************************************************
23: func_62     PROC
24:             push    bx
25:             mov     ah, 62h                 ;Get PSP
26:             int     21h
27:             mov     dx, bx
28:             xor     ax, ax
29:             pop     bx
30:             ret
31: func_62     ENDP
32:
33: ;*******************************************************************
34: ;
35: ;       ルーチン名:   func_48
36: ;       機   能:   ファンクション 48Hの実行
37: ;       func   :   48H（メモリ・ブロックの割り当て）
38: ;       入   力:   arg1 ･･･ ＰＳＰへのポインタ
39: ;                  arg2 ･･･ 必要な割り当てサイズ（パラグラフ）
40: ;       出   力:   DX   ･･･ 割り当てられたメモリの先頭のセグメント・アドレス
41: ;                  AX       0000H
42: ;
43: ;*******************************************************************
44: func_48         PROC    arg1:FAR PTR, arg2:WORD
45:                 push    bx
46:                 push    es
47:                 les     di, arg1
48:                 mov     dx, es                  ;戻り値（エラーの場合）
49:                 mov     bx, arg2
50:                 mov     ah, 48h
51:                 int     21h                     ;ファンクション 48H
52:                 mov     dx, ax                  ;エラーなし
53:                 jnc     noerr
54:                 xor     dx, dx
55: noerr:
56:                 xor     ax, ax
57:                 pop     es
58:                 pop     bx
59:                 ret
60: func_48         ENDP
61:
62: ;*******************************************************************
63: ;
64: ;       ルーチン名 func_49
65: ;       機   能   ファンクション 49Hの実行
66: ;       func      49H（メモリブ・ロックの解放）
67: ;       入   力   arg1   ESレジスタに設定するセグメント・アドレス
68: ;       出   力   DX     ESレジスタ内容
69: ;                 AX     0000H
70: ;
71: ;*******************************************************************
72: func_49         PROC    arg1:FAR PTR
73:                 push    bx
74:                 push    es
75:                 les     di, arg1
76:                 xor     dx, dx                  ;戻り値（エラーの場合）
77:                 mov     ax, es
78:                 inc     ax
79:                 mov     es, ax                  ;つぎのセグメント
```

〔リスト7-24〕
プログラム
malocsub.asm ②

```
 80:            mov     ah, 49h
 81:            int     21h                         ;ファンクション 49H
 82:            mov     dx, es
 83:            pop     es
 84:            jnc     noerr
 85:            xor     dx, dx
 86: noerr:
 87:            xor     ax, ax
 88:            pop     bx
 89:            ret
 90: func_49    ENDP
 91:
 92: ;************************************************************
 93: ;
 94: ;    ルーチン名 :  func_4a
 95: ;    機   能 :  ファンクション 4AH の実行
 96: ;    func       4AH（メモリ・ブロックのサイズ変更）
 97: ;    入   力:   arg1 ··· メモリ領域の先頭アドレス
 98: ;               arg2 ··· 必要な割り当てサイズ（パラグラフ）
 99: ;    出   力:   DX      ESレジスタ内容
100: ;               AX   ··· 0000H
101: ;
102: ;************************************************************
103: func_4a    PROC    arg1:FAR PTR, arg2:WORD
104:            push    bx
105:            push    es
106:            les     di, arg1
107:            mov     dx, es                      ;戻り値
108:            mov     bx, arg2
109:            mov     ah, 4Ah
110:            int     21h                         ;ファンクション 4AH
111:            jnc     noerr
112:            xor     dx, dx                      ;エラー発生
113: noerr:
114:            xor     ax, ax
115:            pop     es
116:            pop     bx
117:            ret
118: func_4a    ENDP
119:
120: ;************************************************************
121: ;
122: ;    ルーチン名 :  add_eseg
123: ;    機   能:   ESレジスタ内容の加算
124: ;    入   力:   arg1 ··· farポインタ
125: ;               arg2 ··· 加算データ
126: ;    出   力:   DX   ESレジスタ内容
127: ;               AX ··· 0000H
128: ;
129: ;************************************************************
130: add_eseg   PROC    arg1:FAR PTR, arg2:WORD
131:            push    es
132:            les     di, arg1
133:            mov     dx, es
134:            add     dx, arg2
135:            mov     ax, di
136:            pop     es
137:            ret
138: add_eseg   ENDP
139:
140: ;************************************************************
141: ;
142: ;    ルーチン名 :  sub_eseg
143: ;    機   能 :  ESレジスタ内容の減算
144: ;    入   力:   arg1 ··· farポインタ
145: ;               arg2 ··· 減算データ
146: ;    出   力:   DX ··· ESレジスタ内容
147: ;               AX ··· 0000H
148: ;
149: ;************************************************************
150: sub_eseg   PROC    arg1:FAR PTR, arg2:WORD
151:            push    es
152:            les     di, arg1
153:            mov     dx, es
154:            sub     dx, arg2
155:            mov     ax, di
156:            pop     es
157:            ret
158: sub_eseg   ENDP
159:            END
```

サブルーチン func_49 は，ファンクション 49H を用いてメモリ・ブロックの解放を行います．このサブルーチンではESレジスタに設定するセグメント・アドレス(arg1)を引数として受け取り，ファンクション 49H を実行します．得られた新しいセグメント・アドレスはFAR ポインタとしてDX：AXレジスタに返します．ここでも，もしファンクション 49H でエラーが発生した場合にはDX：AX レジスタを“0”にして返します．

サブルーチン func_4a はファンクション 4AHを用いてメモリ・ブロックのサイズ変更を行います．このサブルーチンでは，メモリ領域の先頭アドレス(arg1)と必要な割り当てサイズ(arg2)を引数として受け取り，ファンクション 4AH を実行します．得られたセグメント・アドレスの内容はFARポインタとしてDX：AX レジスタに返します．ここでも，もしファンクション 4AH でエラーが発生した場合にはDX：AX レジスタを“0”にして返します．

サブルーチン add_eseg は，FAR ポインタ(セグメント・アドレス)の加算を行います．このサブルーチンでは，FAR ポインタ(arg1)にセグメント値(arg2)を加算し，その結果をFAR ポインタとしてDX：AX レジスタに返します．

サブルーチン sub_eseg は，FAR ポインタ(セグメント・アドレス)の減算を行います．このサブルーチンでは，FAR ポインタ(arg1)からセグメント値(arg2)を減算し，その結果をFAR ポインタとしてDX：AX レジスタに返します．

◆ 生成方法

プログラム maloc.exe は，以下の手順で分割アセンブル/コンパイルして作成します．

```
masm /ML malocsub;
cl −J −As maloc.c malocsub
```

◆ 実行サンプル

リスト7-25は作成したメモリ管理情報表示プログラム maloc.exe の実行例を示しています．

① プログラム maloc.exe を起動する．

② このメモリ・ブロックは，パラグラフの 4A21H から 1184H だけ，すなわち，

4A21H ＋ 1184H ＝ 5BA5H

までのパラグラフ(絶対番地の 5BA50H まで)がプログラム maloc.exe に割り当てられていることになる．

③ 次にファンクション 4AH が実行され，20000H バイトだけのメモリ・ブロックのサイズ変更が行われると，パラグラフ 4A21H からのメモリ・ブロックは 20000H バイトに切り詰められた．

④ メモリ管理情報の第 1 バイトが 5AH となり，このメモリ・ブロックが最後のメモリ・ブロックであることが確認される．

⑤ PSP の先頭アドレスが 0000H となってフリー領域となっていることが確認できる．

⑥ メモリ管理情報のセグメント値 6A21H から 35DEH がフリー領域であり，メモリの上限がパラグラフ 9FFFH であることが確認される．

⑦ ファンクション 48H の実行によってセグメント 4A21H から 2000H バイトのメモリ・ブロックが割り

〔リスト7-25〕プログラム maloc.exe の実行例

```
R>maloc ⏎… プログラムの起動①
 *** メモリ・ブロック表示プログラム Ver.1.1 ***

起動時のメモリ・ブロックの内容
 4A20:0000 4D 21 4A 84 11 A1 16 00 - A3 E3 0B A1 0A 00 A3 E5    M!J.............
                ←このメモリ・ブロックはパラグラフ 4A21+1184=5BA5H まで割り当てられている②
ファンクション 4AH を実行しました.
現時点のメモリ・ブロックの内容 ←このメモリ・ブロックは 20000H バイトの大きさに切り詰められた③
 4A20:0000 4D 21 4A 00 20 A1 16 00 - A3 E3 0B A1 0A 00 A3 E5    M!J. ..........
新たなメモリ・ブロックの内容 ←パラグラフ 6A21+35DE=9FFFH まで RAM が存在⑥
 6A21:0000 5A 00 00 DE 35 0B C0 75 - 04 2B C0 EB 04 0E E8 41    Z...5..u.+.....A
最後のメモリ・ブロック④ ←フリー領域⑤
ファンクション 48H を実行しました.
現時点のメモリ・ブロックの内容 ←空きメモリ・ブロックに 2000H バイトの新たなメモリ・ブロックを割り当てた⑦
 6A21:0000 4D 21 4A 00 02 0B C0 75 - 04 2B C0 EB 04 0E E8 41    M!J....u.+.....A
新たなメモリ・ブロックの内容
 6C22:0000 5A 00 00 DD 33 5E 5F 8B - E5 5D CB 55 8B EC 56 8B    Z...3^_..].U..V.
                ←フリー・メモリがパラグラフ 33DDH まで減っている⑧
ファンクション 49H を実行しました.
現時点のメモリ・ブロックの内容 ←このメモリ・ブロックを開放したので 0000H となる⑨
 6A21:0000 4D 00 00 00 02 0B C0 75 - 04 2B C0 EB 04 0E E8 41    M......u.+.....A
前のメモリ・ブロックの内容
 6C22:0000 5A 00 00 DD 33 5E 5F 8B - E5 5D CB 55 8B EC 56 8B    Z...3^_..].U..V.
                ←この部分は変更されない⑩
R>
```

当てられた．

⑧ すると，最後のメモリ管理情報の示すフリー・メモリがパラグラフ 33DDH に減っている．

⑨ 次に，ファンクション 49H の実行によって ⑦ で得られたメモリ・ブロックを解放すると，そのメモリ・ブロックがフリー・メモリ(0000H)に変わる．

⑩ しかし，最後のメモリ管理情報に変化は生じない．これは，このメモリ領域がファンクション 49H によって解放されたため，すでにメモリ管理情報ではなくなっているからである．

除算エラーを検出する

リスト7-26(divset.asm)は，除算エラー割り込み(INT 00H)のベクタを横取りし，除算エラーが発生した場合にエラー・メッセージとエラーの発生したアドレスを表示する常駐型プログラム divset.com のアセンブリ・ソース・リストです．

● divset.asm

同リストにおいて，プロシージャ div_set は，プロ

［リスト7-26］
プログラム
divset.asm ①

```
 1: ;*************************************************************************
 2: ;
 3: ;     機    能  :  除算エラー割り込みのベクタを横取りする
 4: ;     ファンクション :  02H（文字の出力）
 5: ;                   09H（文字列の出力）
 6: ;                   25H（割り込みベクタの設定）
 7: ;                   31H（プロセスの常駐終了）
 8: ;     生    成  :  masm /ML divset;
 9: ;                   link /NOI divset;
10: ;                   exe2bin divset divset.com
11: ;
12: ;*************************************************************************
13: CODE           SEGMENT
14:                ASSUME  CS:CODE, DS:CODE
15: ;*************************************************************************
16: ;
17: ;       ルーチン名  :  div_set
18: ;       機    能  :  除算エラー処理ルーチンのベクタ設定
19: ;       func      :  09H（文字列の出力）
20: ;                    25H（割り込みベクタの設定）
21: ;                    31H（プロセスの常駐終了）
22: ;       入    力  :  なし
23: ;       出    力  :  AL ･･･ 終了コード00H（エラーなし常駐終了）
24: ;
25: ;*************************************************************************
26:                ORG     100h
27:
28: div_set        PROC    NEAR
29:                push    cs
30:                pop     ds
31:                lea     dx, open_msg
32:                mov     ah, 09h
33:                int     21h                     ;プログラム・タイトル表示
34:                lea     dx, int_0
35:                mov     ah, 25h
36:                mov     al, 00h
37:                int     21h                     ;ベクタ・セット
38:                lea     dx, tail
39:                mov     cl, 4
40:                shr     dx, cl                  ;常駐パラグラフ
41:                inc     dx
42:                mov     ah, 31h
43:                xor     al, al
44:                int     21h                     ;常駐終了
45: div_set        ENDP
46:
47: ;*************************************************************************
48: ;
49: ;       ルーチン名  :  int_0
50: ;       機    能  :  除算エラーの発生アドレスを表示
51: ;       func      :  09H（文字列の出力）
52: ;       入    力  :  なし
53: ;       出    力  :  なし
54: ;
55: ;*************************************************************************
56: int_0          PROC    FAR
57:                push    ax
58:                push    cx
59:                push    dx
```

グラム・タイトルを表示したのち，ファンクション25Hを用いてINT 00H(除算エラー割り込み)に対してプロシージャint_0のエントリ・アドレスを登録します．

そして，プログラムdivset.comの最終セグメントの計算を行ってDXレジスタに設定し，ファンクション31Hによってプログラムdivset.comをメモリに残したまま常駐終了します．

そのあとに，ユーザ・プログラム内で除算エラー(0で除算)が発生すると，INT 00Hの割り込みが発生しプロシージャint_0に制御が移ります．プロシージャint_0では，すべてのレジスタを退避したのちエラー・メッセージを表示し，そのあとにスタックに積まれている戻り番地を参照してエラーの発生したアドレスの表示を行います．

プロシージャput_axはAXレジスタの内容を16進表示します．同様にプロシージャput_alはALレジスタの内容を16進表示し，プロシージャput_1はALレジスタ下位4ビットをASCIIコードに変換して16進数で表示します．

〔リスト7-26〕
プログラム
divset.asm ②

```
 60:            push    bp
 61:            push    ds
 62:
 63:            mov     bp, sp
 64:            add     bp, 10
 65:            push    cs
 66:            pop     ds
 67:            lea     dx, err_msg1
 68:            mov     ah, 09h
 69:            int     21h                     ;エラー・メッセージ
 70:            mov     ax, [bp + 2]            ;セグメント
 71:            call    put_ax
 72:            lea     dx, err_msg2
 73:            mov     ah, 09h
 74:            int     21h                     ;コロン表示
 75:            mov     ax, [bp + 0]            ;オフセット
 76:            call    put_ax
 77:            lea     dx, err_msg3
 78:            mov     ah, 09h
 79:            int     21h
 80:            pop     ds
 81:            pop     bp
 82:            pop     dx
 83:            pop     cx
 84:            pop     ax
 85:            iret
 86: int_0      ENDP
 87:
 88: ;*******************************************************************
 89: ;
 90: ;    ルーチン名：  put_ax
 91: ;    機    能：  AXの内容を表示
 92: ;    入    力：  AX … 16進数（2バイト）
 93: ;    出    力：  なし
 94: ;
 95: ;*******************************************************************
 96: put_ax     PROC
 97:            push    ax
 98:            push    cx
 99:
100:            push    ax
101:            mov     al, ah
102:            call    put_al                  ;上位バイト
103:            pop     ax
104:            call    put_al                  ;下位バイト
105:
106:            pop     cx
107:            pop     ax
108:            ret
109: put_ax     ENDP
110:
111: ;*******************************************************************
112: ;
113: ;    ルーチン名：  put_al
114: ;    機    能：  ALレジスタの表示
115: ;    入    力：  AL … 16進数（1バイト）
116: ;    出    力：  なし
117: ;
118: ;*******************************************************************
```

〔リスト7-26〕
プログラム
divset.asm ③

```
119: put_al    PROC
120:           push    ax
121:           push    cx
122:
123:           push    ax
124:           mov     cl, 4
125:           shr     al, cl
126:           call    put1
127:           pop     ax
128:           call    put1
129:
130:           pop     cx
131:           pop     ax
132:           ret
133: put_al    ENDP
134:
135: ;*****************************************************************
136: ;
137: ;     ﾙｰﾁﾝ名 :   put_1
138: ;     機  能 :   1桁16進数を表示
139: ;     func   :   02H(文字の出力)
140: ;     入  力 :   AL    16進数(1桁)
141: ;     出  力 :   なし
142: ;
143: ;*****************************************************************
144: put1      PROC
145:           push    ax
146:           push    dx
147:           and     al, 0Fh
148:           cmp     al, 0Ah
149:           jb      num
150:           add     al, 07h
151: num:
152:           add     al, '0'
153:           mov     dl, al
154:           mov     ah, 02h
155:           int     21h
156:           pop     dx
157:           pop     ax
158:           ret
159: put1      ENDP
160:
161: ;*****************************************************************
162: ;
163: ;     ﾃﾞｰﾀ名 :   open_msg
164: ;     機  能 :   プログラム・タイトル
165: ;
166: ;     ﾃﾞｰﾀ名 :   err_msg(n)
167: ;     機  能 :   エラー・メッセージ
168: ;
169: ;*****************************************************************
170: open_msg  DB      0Dh, 0Ah, '除算エラー処理ルーチンが常駐しました.'
171:           DB      0Dh, 0Ah, '$'
172: err_msg1  DB      0Dh, 0Ah, 07h,'プログラムの内部エラーです.'
173:           DB      0Dh, 0Ah 'アドレス', '$'
174: err_msg2  DB      ':', '$'
175: err_msg3  DB      'において0で除算をしました.'
176:           DB      0Dh, 0Ah,'$'
177: tail      LABEL   NEAR
178: CODE      ENDS
179:           END     div_set
```

● div0.asm

リスト7-27(div0.asm)は，除算エラーを発生するプログラムdiv0.exeのソース・プログラムです．

同リストにおいて，プロシージャdiv0はプログラム・タイトルの表示を行ったのち，BLレジスタに“0”を設定して故意にBLレジスタでの除算を行って除算エラーを発生させています．

そのあとは，除算エラー処理ルーチン(**リスト7-26**のプロシージャint_0)から正常に制御が戻ったことを確認するために，プログラム終了の表示を行ってファンクション4CHを用いてプログラムdiv0.exeを正常終了します．

この除算エラーを発生するプログラムは，C言語で記述することも可能なのですが，MS-Cではコンパイルされたプログラムの実行時に独自の除算エラー処理ルーチンを登録するため，このようにアセンブリ言語で記述しました．

◆ 生成方法

プログラムdivset.comは，次の手順でアセンブル/リンクして作成します．

〔リスト7-27〕
プログラム
div0.asm

```
 1: ;************************************************************
 2: ;
 3: ;    機    能 :   除算エラーを発生する
 4: ;    ファンクション :   09H ( 文字列の出力 )
 5: ;                4CH ( プロセスの終了 )
 6: ;    生    成 :   masm /ML div0;
 7: ;                link /NOI div0;
 8: ;
 9: ;************************************************************
10:         .MODEL   SMALL
11:         .STACK   100h
12:         .CODE
13: ;************************************************************
14: ;
15: ;     ルーチン名 :   div0
16: ;     機    能 :   除算エラーの発生
17: ;     func     :   09H ( 文字列の出力 )
18: ;                 4CH ( プロセスの終了 )
19: ;     入    力 :   なし
20: ;     出    力 :   AL ・・・ 終了コード ( 00H = エラーなし )
21: ;
22: ;************************************************************
23: div0       PROC     NEAR
24:            push     cs
25:            pop      ds
26:            lea      dx, open_msg
27:            mov      ah, 09h
28:            int      21h                   ;プログラム・タイトル表示
29:            mov      bl, 0
30:            div      bl                    ;0で除算
31:            lea      dx, close_msg
32:            mov      ah, 09h
33:            int      21h                   ;プログラム終了表示
34:            mov      ah, 4Ch
35:            xor      al, al
36:            int      21h                   ;プログラム終了
37: div0       ENDP
38:
39: ;************************************************************
40: ;
41: ;     データ名 :   open_msg
42: ;     機    能 :   プログラム・タイトル
43: ;
44: ;     データ名 :   close_msg
45: ;     機    能 :   終了メッセージ
46: ;
47: ;************************************************************
48: open_msg     DB     0Dh, 0Ah, '除算エラー発生プログラム Ver.1.1'
49:              DB     0Dh, 0Ah, 0Dh, 0Ah, '$'
50: close_msg    DB     0Dh, 0Ah, '除算エラープログラムを終了します.'
51:              DB     0Dh, 0Ah, '$'
52:              END    div0
```

masm /ML divset;
link /NOI divset;
exe2bin divset divset.com

プログラム divset.com は，プログラム・サイズが小さく，メモリに常駐するため，COM モデルとして作成します.

プログラム div0.exe は，次の手順でアセンブル/リンクして作成します.

masm /ML div0;
link /NOI div0;
(div0.exe は COM モデルにしない)

◆ 実行サンプル

リスト7-28 は，プログラム divset.com および div0.exe の実行例を示しています.

① まず，プログラム divset.com を起動して除算エラー処理ルーチンをメモリ内に常駐させる.

② ここで，div0.exe を起動し除算エラーを発生させる.

③ div0.exe のプログラム・タイトルが表示され，div0.exe が正常に起動されたことが確認できる.

④ 除算エラーが発生し，INT 00H 割り込みが実行される. INT 00H 割り込みでは，除算エラー処理ルーチン(プロシージャ int_0)に制御が移り，そのエラー・メッセージとエラーの発生したアドレスの表示が行われる

⑤ 除算エラー処理ルーチン(int_0)での処理が終わると，除算エラーを発生したプログラム div0.exe に制御が戻り，プログラム div0.exe の終了メッセージが表

〔リスト7-28〕
プログラムdivset.comとdiv0.exeの実行例

```
R>divset ⏎… プログラムの起動(メモリに常駐)①

除算エラー処理ルーチンが常駐しました.

R>div0 ⏎… プログラムの起動②

除算エラー発生プログラム Ver.1.1        ← div0.exeからのメッセージ③

プログラムの内部エラーです.
アドレス 4A60:000E において0で除算をしました.   ← 除算エラーの発生④

除算エラープログラムを終了します.        ← div0.exeに処理が戻っている⑤

R>
```

示されて正常に復帰したことが確認できる.

割り込みエントリ・アドレスを表示する

リスト7-29(gint.c)および**リスト7-30**(gintsub.asm)は，指定されたベクタ番号に対応する割り込みエントリ・アドレスを読み出して表示するプログラムgint.exeのソース・リストです.

● gint.c

リスト7-29はプログラムgint.exeのCソース部分です.同リストにおいて，関数mainでは最初にプログラム・タイトルを表示したのち，コマンド・ライン・パラメータの解析を行います.

パラメータの解析では，まずパラメータの個数をチェックし，もしベクタ番号が指定されていなければプログラムgint.exeの使用方法を表示してエラー・スト

● MS-DOS標準のコントロール・キャラクタ ●

MS-DOSでは，ctrlキーと同時に各種のキーを押すことによって，いろいろな機能を実行することができます.この各種のキー入力をコントロール・キャラクタと呼んでいます.**表B**は，MS-DOS標準のコントロール・キャラクタの一覧です.

〔表B〕
標準コントロール・キャラクタ

操作	機能
ctrl+C	現在実行中のプロセスを中断する
ctrl+H	コマンド・ラインの最終文字を除去し，画面から文字を消去する(BSキーと等価)
ctrl+J	コマンド・ラインの継続.論理行を拡張する
ctrl+P	プリンタへのエコー出力の開始/終了(トグル)
ctrl+N	ctrl+Pと等価
ctrl+S	画面出力の一時中止(任意のキー・インにより再開)
ctrl+X	コマンド・ラインの取り消しを行い，新しいコマンドの入力を可能にする

〔リスト7-29〕
プログラム gint.c

```
 1: /*************************************************************************
 2: *
 3: *    機    能：   ベクタ・アドレスの表示
 4: *    サ    ブ：   gintsub.asm
 5: *    生    成：   masm /ML gintsub;
 6: *                 cl -J -AS gint.c gintsub
 7: *    使用方法：   gint <ベクタ番号>
 8: *
 9: *************************************************************************/
10: #include     <stdio.h>
11: #include     <string.h>
12: #include     <dos.h>
13:
14: void main (int, char **);
15: void int_msg (int, int, int);
16: char far *func_35 (int);              /* ベクタ・アドレスの取得 */
17: /*************************************************************************
18: *
19: *    関数名：    main (argc, argv)
20: *    機  能：    ベクタ番号の読み出し
21: *    入  力：    int argc       ･･･  コマンドライン・パラメータの数
22: *                char *argv[]   ･･･  パラメータ文字列へのポインタ
23: *    出  力：    なし
24: *
25: *************************************************************************/
26: void main (argc, argv)
27: int argc;
28: char **argv;
29: {
30:     char far *ptr;
31:     int n;
32:
33:     printf ("¥n *** 割り込みベクタ表示プログラム Ver.1.1 ***¥n¥n");
34:     if (argc != 2) {
35:         printf ("使用法：gint ベクタ番号¥n¥n");
36:         exit (1);
37:     }
38:     sscanf (*++argv, "%x", &n);
39:     ptr = func_35 (n);
40:     int_msg (n, FP_SEG (ptr), FP_OFF(ptr));
41:     printf ("¥n");
42:     exit (0);
43: }
44:
45: /*************************************************************************
46: *
47: *    関数名：    int_msg (n, seg, off)
48: *    機  能：    ベクタ番号と割り込みアドレスの表示
49: *    入  力：    n    ･･･ ベクタ番号
50: *                seg  ･･･ セグメント・アドレス
51: *                off  ･･･ オフセット・アドレス
52: *    出  力：    なし
53: *
54: *************************************************************************/
55: void int_msg (n, seg, off)
56: int n, seg, off;
57: {
58:     printf ("ベクタ番号          ･･･   %04X (%d)¥n", n, n);
59:     printf ("割り込みアドレス    ･･･   %04X:%04X¥n", seg, off);
60: }
```

ップします．コマンド・ライン・パラメータにベクタ番号が指定されていれば，その番号を16進数とみなしてライブラリ関数sscanfを用いて変数nに格納します．

次に，関数(サブルーチン)func_35を用いて，指定されたベクタ番号に対応した割り込みエントリ・アドレスを読み出し，その結果をFARポインタとして受け取ります．そして，その返されたFARポインタ(割り込みエントリ・アドレス)をセグメント部とオフセット部に分解し，関数int_msgに渡してアドレスの表示を行います．

関数int_msgでは，引数として渡されたベクタ番号と，割り込みエントリ・アドレスを表示します．ここで割り込みエントリ・アドレスは，セグメント：オフセットの形式で表示します．

● gintsub.asm

リスト7-30はプログラムgint.exeのアセンブリ・ソース部分です．同リストにおいて，サブルーチン(関数)func_35は，ファンクション35Hを用いてarg1で指

〔リスト7-30〕
プログラム
gintsub.asm

```
 1: ;****************************************************************
 2: ;
 3: ;    機    能 : 割り込みベクタ取得サブルーチン
 4: ;   ファンクション : 35H(ベクタ アドレスの取得)
 5: ;    生    成 : masm /ML gintsub;
 6: ;
 7: ;****************************************************************
 8:           .MODEL  SMALL, C
 9:           CODE
10: ;****************************************************************
11: ;
12: ;    ルーチン名 :  func_35
13: ;    機    能 :  ベクタ・アドレスを返す
14: ;    func    :  35H(ベクタ アドレスの取得)
15: ;    入    力 :  arg1 ... ベクタ番号
16: ;    出    力 :  DX:AX  ... 割り込み処理アドレス
17: ;
18: ;****************************************************************
19: func_35       PROC    arg1:PTR
20:               push    es
21:               mov     ax, arg1
22:               mov     ah, 35h                     ;ファンクション35H
23:               int     21h
24:               mov     dx, es
25:               mov     ax, bx
26:               pop     es
27:               ret
28: func_35       ENDP
29:               END
```

〔リスト7-31〕プログラム gint.exe の実行例

```
R>gint 5 ⏎… ベクタ番号5を指定してプログラムを起動①

 *** 割り込みベクタ表示プログラム Ver.1.1 ***

ベクタ番号          ....  0005 (5)
割り込みアドレス    ....  0FDB:0853 ◀─(割り込み処理アドレスに注目②)

R>tm ⏎… 時間表示プログラム(tm.com)の起動. COPY キー入力に対応した INT 05H のベクタが書き換えられる③

時間表示プログラムが常駐しました.
時間を表示する際には COPYキーを押して下さい

R>gint 5 ⏎… 再びベクタ番号5を指定してプログラムを起動④

 *** 割り込みベクタ表示プログラム Ver.1.1 ***

ベクタ番号          ....  0005 (5)
割り込みアドレス    ....  4B06:0123 ◀─(割り込み処理アドレスが変更された⑤)

R>
```

定されたベクタ番号に対応した割り込みエントリ・アドレスを読み出し，そのアドレスを FAR アドレスとして DX：AX レジスタに返します．

◆ **生成方法**

プログラム gint.exe は，次の手順で分割アセンブル/コンパイルして作成します．

masm /ML gintsub;

cl －J －As gint.c gintsub

◆ **実行サンプル**

リスト7-31 は，割り込みエントリ・アドレス表示プログラム gint.exe の実行例を示しています．

① まず，パラメータとしてベクタ番号に5を指定してプログラム gint.exe を起動する．

② ここで，表示された割り込みエントリ・アドレス 0FDB：0853 に注目．

③ 次に，すでに紹介した時刻表示プログラム tm.com を起動する．プログラム tm.com では，COPY キーの機能を横取りするため，ベクタ番号5の割り込みエントリ・アドレスを書き替える．

④ 再び割り込みエントリ・アドレス表示プログラム gint.exe に対し，ベクタ番号5を指定して起動する．

⑤ ここで割り込みエントリ・アドレスが 4B06：0123 に変更されている．

7-4 ディスク/ファイルの操作

ディスクのリセット/カレント・ドライブの変更を行う

リスト7-32(dres.c)および**リスト7-33**(dressub.asm)は，ディスクのリセット(ディスク・バッファのフラッシュ)およびカレント・ドライブの変更を行うためのテスト・プログラムdres.exeのソース・リストです．

また，同プログラムでは，そのシステムで定義されている論理ドライブ数(config.sysファイル内のLASTDRIVEコマンドで指定)の表示も行います．

● dres.c

リスト7-32はプログラムdres.exeのCソース部分です．同リストにおいて，関数mainでは最初にプログラム・タイトルを表示したのち，コマンド・ライン・パラメータの解析を行います．

パラメータの解析では，もしコマンド・ラインでドライブの指定(カレント・ドライブの変更)がされていれば，そのドライブ名をドライブ番号に変換します．もし，ドライブの指定がなければ関数(サブルーチン)func_19を用いてカレント・ドライブ番号の読み出しを行い，そのドライブ番号を変数drv_numに格納します．

これらのパラメータの解析が終わったら，関数func_0eにドライブ番号drv_numを渡してカレント・ドライブの選択を行います．そして，関数func_0eからは論理ドライブ数が返されるので，その論理ドライブ数(ドライブ名)の表示を行ったのち，関数func_0dを用いてディスクのリセットを行い，ディスク・バッファをフラッシュします．

● dressub.asm

リスト7-33はプログラムdres.exeのアセンブリ・ソース部分です．同リストにおいて，サブルーチン(関数)func_19は，ファンクション19Hを用いてカレント・ドライブ番号の読み出しを行い，その読み出されたカレント・ドライブ番号をAXレジスタに返しま

〔リスト7-32〕プログラムdres.c

```
 1: /**********************************************************************
 2: *
 3: *   機    能 :  ディスクのリセットとカレント・ドライブの変更
 4: *   サ    ブ :  dressub.asm
 5: *   生    成 :  masm /ML dressub;
 6: *               cl -J -AS dres.c dressub
 7: *   使用方法 :  dres [<d:>]
 8: *
 9: **********************************************************************/
10: #include    <stdio.h>
11:
12: void main (int, char **);
13: void func_0d (void);                    /* ディスクのリセット */
14: int func_0e (int);                      /* ドライブの選択 */
15: int func_19 (void);                     /* カレント・ドライブ番号の取得 */
16: /**********************************************************************
17: *
18: *   関数名 :  main (argc, argv)
19: *   機 能 :
20: *   入 力 :  int argc        .....  コマンドライン・パラメータの数
21: *            char *argv[]    .....  パラメータ文字列へのポインタ
22: *   出 力 :  なし
23: *
24: **********************************************************************/
25: void main (argc, argv)
26: int argc;
27: char **argv;
28: {
29:     int drv_num;
30:
31:     printf ("¥n *** ディスク・リセット・プログラム Ver.1.1 ***¥n¥n");
32:     drv_num = 0;
33:     if (argc != 1) {
34:         drv_num = toupper (**++argv) - 'A';
35:     } else {
36:         drv_num = func_19 ();
37:     }
38:     drv_num = func_0e (drv_num);
39:     printf ("論理ドライブ数 = %d 台 (%cドライブ) ¥n", drv_num, drv_num + '@');
40:     func_0d ();
41:     printf ("¥n");
42:     exit (0);
43: }
```

〔リスト7-33〕
プログラム
dressub.asm

```
 1: ;*****************************************************************
 2: ;
 3: ;     機     能：   ディスク管理サブルーチン
 4: ;   ファンクション   0DH（ディスクのリセット）
 5: ;                   0EH（ドライブの選択）
 6: ;                   19H（カレント・ドライブ番号の呼び出し）
 7: ;     生     成：   masm /ML dressub;
 8: ;
 9: ;*****************************************************************
10:                 .MODEL   SMALL, C
11:                 .CODE
12: ;*****************************************************************
13: ;
14: ;     ルーチン名：  func_19
15: ;     機   能：    カレント・ドライブ番号の取得
16: ;     func   ：    19H（カレント・ドライブ番号の読み出し）
17: ;     入   力：    なし
18: ;     出   力：    AX ･･･ ドライブ番号（00H=A, 01H=B, ･･･）
19: ;
20: ;*****************************************************************
21: func_19         PROC
22:                 mov      ah, 19h                ;ファンクション19H
23:                 int      21h
24:                 xor      ah, ah
25:                 ret
26: func_19         ENDP
27:
28: ;*****************************************************************
29: ;
30: ;     ルーチン名：  func_0e
31: ;     機   能：    ドライブの選択
32: ;     func   ：    0EH（ドライブの選択）
33: ;     入   力：    arg1 ･･･ ドライブ番号
34: ;     出   力：    AX   ･･･ 論理ドライブ数
35: ;
36: ;*****************************************************************
37: func_0e         PROC     arg1:WORD
38:                 mov      dx, arg1
39:                 mov      ah, 0Eh
40:                 int      21h
41:                 xor      ah, ah
42:                 ret
43: func_0e         ENDP
44:
45: ;*****************************************************************
46: ;
47: ;     ルーチン名：  func_0d
48: ;     機   能：    ディスク・リセット
49: ;     func   ：    0DH（ディスクのリセット）
50: ;     入   力：    なし
51: ;     出   力：    なし
52: ;
53: ;*****************************************************************
54: func_0d         PROC
55:                 mov      ah, 0Dh
56:                 int      21h
57:                 ret
58: func_0d         ENDP
59:                 END
```

す．

サブルーチンfunc_0eは，ファンクション0EHを用いて引数arg1で指定されたドライブ番号のドライブをカレント・ドライブに変更します．

サブルーチンfunc_0dは，ファンクション0DHを用いてディスクのリセットを行い，ディスク・バッファのフラッシュを行います．

◆ **生成方法**

プログラムdres.exeは，次の手順で分割アセンブル/コンパイルして作成します．

masm /ML dressub;

cl −J −As dres.c dressub

◆ **実行サンプル**

リスト7-34はディスク管理のテスト・プログラムdres.exeの実行例を示しています．

① まず，dirコマンドを用いてドライブBをアクセスする．

② 再びdirコマンドを用いてドライブBをアクセスする．すると，ここではディスク・バッファの効果で，実際のディスクへのアクセスは行われずバッファの内容が表示されることになる．

③ ここでプログラムdres.exeを実行する．これによ

〔リスト7-34〕プログラム dres.exe の実行例

```
R>dir b: ⏎… ドライブBのアクセス①

 ドライブ B: のディスクのボリュームラベルはありません.
 ディレクトリは B:¥

PRO        <DIR>      88-12-06  16:17
      1 個のファイルがあります.
  1080320 バイトが使用可能です.

R>dir b: ⏎… 再びドライブBのアクセス②(ドアを閉めているのでディスクはアクセスされない)
 ドライブ B: のディスクのボリュームラベルはありません.
 ディレクトリは B:¥

PRO        <DIR>      88-12-06  16:17
      1 個のファイルがあります.
  1080320 バイトが使用可能です.

R>dres ⏎… プログラムの起動③

 *** ディスク・リセット・プログラム Ver.1.1 ***

論理ドライブ数 = 26 台(Zドライブ)  (config.sys ファイルで LASTDRIVE コマンドで指定した論理ドライブ④)
R>dir b: ⏎… 再びドライブBのアクセス(ドアは閉めたまま)⑤

 ドライブ B: のディスクのボリュームラベルはありません.
 ディレクトリは B:¥

PRO        <DIR>      88-12-06  16:17
      1 個のファイルがあります.
  1080320 バイトが使用可能です.

R>dres b: ⏎… パラメータを指定する⑥

 *** ディスク・リセット・プログラム Ver.1.1 ***

論理ドライブ数 = 26 台(Zドライブ)

B>◀─(カレント・ドライブが変更される⑦)
```

ってディスク・バッファはフラッシュされる.

④ プログラム dres.exe では，そのシステムで定義されている論理ドライブ数が表示される．この場合は，config.sys ファイル内に LASTDRIVE コマンドを用いて指定したドライブ名 "Z" が表示される.

⑤ 再び dir コマンドを実行する．ここでは，プログラム dres.exe によってディスク・バッファがフラッシュされているので，実際にディスクへのアクセスが行われる.

⑥ プログラム dres.exe にドライブ名を指定して起動する.

⑦ すると，カレント・ドライブが指定したドライブに変更される.

ディスク情報を得る

リスト7-35(getd.c)および**リスト7-36**(getdsub.asm)は，指定されたドライブのセクタ数やクラスタ数などディスクに関する情報を表示するプログラム getd.exe のソース・リストです.

● getd.c

リスト7-35 はプログラム getd.exe の C ソース部分です．同リストにおいて，構造体 _DSK はファンクション 36H から各レジスタに返されるディスク情報を格納するデータ領域の構造を定義します.

関数 main では，最初にプログラム・タイトルを表示したのち，コマンド・ライン・パラメータの解析を行います．パラメータの解析では，コマンド・ラインでドライブ名が指定されていれば，そのドライブ名をドライブ番号に変換します．もし，ドライブ名が省略されていれば，関数(サブルーチン)func_19 を用いてカレント・ドライブのドライブ番号を取得して変数 drv_num に格納しておきます.

これらのコマンド・ラインのパラメータ解析が終わったら，次に関数 dsk_read にそのドライブ番号 drv_num を引数として渡して，ディスク情報の取得と表示を行います.

関数 dsk_read では，まずディスク情報を格納するための構造体宣言を行って，そのデータ領域を確保します．次に，関数 func_36 に指定されたドライブ番号と構造体へのポインタを渡し，ディスク情報を構造体中の各メンバに読み出します．

ここで，ドライブの指定が間違っているなどのエラーが発生すると，構造体中のメンバ ax に 0FFFFH が返されるので，その場合はエラー情報を表示してプログラム dres.exe をエラー・ストップします．もし，エラーがなければ関数 func_1c にドライブ番号を渡し，ディスクの FAT-ID へのポインタを読み出して構造体中のメンバ idptr に格納します．

これらの処理によって，すべてのディスク情報が得られたことになるので，関数 disk_msg にドライブ番号と構造体へのポインタを渡してディスク情報の表示を行います．

関数 disk_msg では，指定されたドライブ番号およびディスク情報の格納されている構造体へのポインタから，あらかじめディスク容量や残り容量の計算を行ったり，FAT-ID へのポインタから FAT-ID を読み込んで各変数に設定し，そのあとでディスクの各情報の表示を行います．

〔リスト7-35〕プログラム getd.c ①

```
 1: /***************************************************************
 2: *
 3: *   機    能 :  ディスク・データの表示
 4: *   サ    ブ :  getdsub.asm
 5: *   生    成 :  masm /ML getdsub;
 6: *               cl -J -AS getd.c getdsub
 7: *   使用方法 :  getd [<d:>]
 8: *
 9: ***************************************************************/
10: #include    <stdio.h>
11:
12: /***************************************************************
13: *
14: *    構造体 :   _DSK
15: *    機  能 :   ディスク・データ構造の定義
16: *
17: ***************************************************************/
18: typedef struct _DSK {
19:     unsigned int ax;          /* 1クラスタあたりのセクタ数 */
20:     unsigned int bx;          /* 使用可能なクラスタ数 */
21:     unsigned int cx;          /* 1セクタあたりのバイト数 */
22:     unsigned int dx;          /* 1ドライブあたりのクラスタ数 */
23:     char far *idptr;          /* FAT ID へのポインタ */
24: } DSK;
25:
26: void main (int, char **);
27: void dsk_read (int);
28: void disk_msg (int, DSK *);
29: int  func_19 (void);              /* カレント・ドライブ番号の読み出し */
30: void func_36 (int, DSK *);  /* ディスク残り容量の読み出し */
31: char far *func_1c (int);          /* FATアドレスの読み出し */
32: /***************************************************************
33: *
34: *    関数名 :   main (argc, argv)
35: *    機  能 :   コマンドライン・パラメータの解析
36: *    入  力 :   int argc        ....  コマンドライン・パラメータの数
37: *               char *argv[]    ....  パラメータ文字列へのポインタ
38: *    出  力 :   なし
39: *
40: ***************************************************************/
41: void main (argc, argv)
42: int argc;
43: char **argv;
44: {
45:     int drv_num;
46:
47:     printf ("¥n *** ディスク情報表示プログラム Ver.1.1 ***¥n¥n");
48:     drv_num = 0;
49:     if (argc != 1) {
50:         drv_num = toupper (**++argv) - 'A';
51:     } else {
52:         drv_num = func_19 ();
53:     }
54:     dsk_read (drv_num);
55:     printf ("¥n");
56:     exit (0);
```

〔リスト7-35〕プログラム getd.c ②

```
 57: }
 58:
 59: /**********************************************************************
 60: *
 61: *   関数名 :   dsk_read (drv_num)
 62: *   機  能 :   ディスク情報の取得と表示
 63: *   入  力 :   drv_num ‥‥ ドライブ番号
 64: *   出  力 :   なし
 65: *
 66: **********************************************************************/
 67: void dsk_read (drv_num)
 68: int drv_num;
 69: {
 70:     DSK disk_data;
 71:
 72:     func_36 (drv_num + 1, &disk_data);
 73:     if (disk_data.ax == 0xFFFF) {
 74:         printf ("ドライブの指定が無効です.¥n");
 75:         exit (1);
 76:     }
 77:     disk_data.idptr = func_1c (drv_num + 1);
 78:     disk_msg (drv_num, &disk_data);
 79: }
 80:
 81: /**********************************************************************
 82: *
 83: *   関数名 :   disk_msg (drv_num, ptr)
 84: *   機  能 :   ディスク情報の表示
 85: *   入  力 :   drv_num ‥ ドライブ番号
 86: *              ptr     ‥ ディスク・データ構造体へのポインタ
 87: *   出  力 :   なし
 88: *
 89: **********************************************************************/
 90: void disk_msg (drv_num, ptr)
 91: DSK *ptr;
 92: int drv_num;
 93: {
 94:     long bytes0, bytes1;
 95:     int c;
 96:
 97:     bytes1 = (long)ptr -> bx * ptr -> ax * ptr -> cx;
 98:     bytes0 = (long)ptr -> dx * ptr -> ax * ptr -> cx;
 99:     c = *ptr -> idptr;
100:     printf ("ドライブ %c のディスク情報¥n¥n", (char)(drv_num + 'A'));
101:     printf ("FAT ID                       ---   %2X¥n", c & 0xFF);
102:     printf ("1ドライブあたりのクラスタ数   ---   %d クラスタ¥n", ptr -> dx);
103:     printf ("1クラスタあたりのセクタ数     ---   %d セクタ¥n", ptr -> ax);
104:     printf ("1セクタあたりのバイト数       ---   %d バイト¥n", ptr -> cx);
105:     printf ("ディスク容量                  ---   %ld バイト¥n", bytes0);
106:     printf ("残り容量                      ---   %ld バイト¥n", bytes1);
107: }
```

〔リスト7-36〕
プログラム
getdsub.asm ①

```
  1: ;**********************************************************************
  2: ;
  3: ;   機     能 :   ディスク情報読み出しサブルーチン
  4: ;   ファンクション :   19H (カレント・ドライブ番号の読み出し)
  5: ;                1CH (指定ドライブデータの取得)
  6: ;                36H (ディスク情報の読み出し)
  7: ;   生     成 :   masm /ML detdsub;
  8: ;
  9: ;**********************************************************************
 10:               .MODEL  SMALL, C
 11:               .CODE
 12: ;**********************************************************************
 13: ;
 14: ;     構造体 :    _DSK
 15: ;     機  能 :    ディスク・データ構造の定義
 16: ;
 17: ;**********************************************************************/
 18: _DSK          STRUC
 19: ax_data       DW      ?
 20: bx_data       DW      ?
 21: cx_data       DW      ?
 22: dx_data       DW      ?
```

〔リスト7-36〕
プログラム
getdsub.asm ②

```
23: idptr          DD         ?
24: _DSK           ENDS
25:
26: ;*************************************************************
27: ;
28: ;     ルーチン名 :    func_19
29: ;     機    能 :    カレント・ドライブ番号の取得
30: ;     func     :    19H(カレント・ドライブ番号の読み出し)
31: ;     入    力 :    なし
32: ;     出    力 :    AX ・・ドライブ番号(00H=A, 01H=B, ・・・)
33: ;
34: ;*************************************************************
35: func_19        PROC
36:                mov        ah, 19h               ;ファンクション19H
37:                int        21h
38:                xor        ah, ah
39:                ret
40: func_19        ENDP
41:
42: ;*************************************************************
43: ;
44: ;     ルーチン名 :    func_36
45: ;     機    能 :    ディスク・データの読み出し
46: ;     func     :    36H(ディスク残り容量の読み出し)
47: ;     入    力 :    arg1 ・・・ドライブ番号
48: ;                   arg2・・・データ構造体へのポインタ
49: ;     出    力 :    なし(構造体の中へ設定)
50: ;
51: ;*************************************************************
52: func_36        PROC       arg1:WORD, arg2:PTR
53:                push       si
54:                mov        dx, arg1              ;ドライブ番号
55:                mov        ah, 36h               ;ファンクション36H
56:                int        21h
57:                mov        si, arg2
58:                mov        [si.ax_data], ax      ;1クラスタあたりのセクタ数
59:                mov        [si.bx_data], bx      ;使用可能なクラスタ数
60:                mov        [si.cx_data], cx      ;1セクタあたりのバイト数
61:                mov        [si.dx_data], dx      ;1ドライブあたりのクラスタ数
62:                pop        si
63:                ret
64: func_36        ENDP
65:
66: ;*************************************************************
67: ;
68: ;     ルーチン名 :    func_1c
69: ;     機    能 :    FAT ID アドレスの取得
70: ;     func     :    1CH(指定ドライブ・データの取得)
71: ;     入    力 :    arg1 ・・・ドライブ番号
72: ;     出    力 :    DX:AX ・・FAT ID アドレス
73: ;
74: ;*************************************************************
75: func_1c        PROC       arg1:WORD
76:                push       si
77:                push       ds
78:                mov        dx, arg1              ;ドライブ番号
79:                mov        ah, 1Ch               ;ファンクション1CH
80:                int        21h
81:                mov        dx, ds
82:                mov        ax, bx
83:                pop        ds
84:                pop        si
85:                ret
86: func_1c        ENDP
87:                END
```

● getdsub.asm

リスト7-36はプログラムgetd.exeのアセンブリ・ソース部分です．同リストにおいて，ストラクチャ_DSKは**リスト7-35**のCソース内における構造体_DSKに対応していて，ファンクション36Hで各レジスタに返されたディスク情報を格納するデータ領域の構造を定義しています．

サブルーチン(関数)func_19は，ファンクション19Hを用いてカレント・ドライブ番号の読み出しを行い，そのドライブ番号をAXレジスタに返します．

サブルーチンfunc_36は，引数arg1で指定された

〔リスト7-37〕プログラム getd.exe の実行例

```
R>getd ⏎…カレント・ドライブの表示①

*** ディスク情報表示プログラム Ver.1.1 ***
                               (Rドライブは RAMディスク)
ドライブ R のディスク情報

FAT ID                        ---  F8                  ←(1.5Mバイトの RAMディスクの情報②)
1ドライブあたりのクラスタ数   ---  760 クラスタ
1クラスタあたりのセクタ数     ---  4 セクタ
1セクタあたりのバイト数       ---  512 バイト
ディスク容量                  ---  1556480 バイト
残り容量                      ---  661504 バイト

R>getd y: ⏎…ドライブYの表示③

*** ディスク情報表示プログラム Ver.1.1 ***

ドライブ Y のディスク情報

FAT ID                        ---  FE                  ←(1Mバイトのフロッピ・ディスク④)
1ドライブあたりのクラスタ数   ---  1221 クラスタ
1クラスタあたりのセクタ数     ---  1 セクタ
1セクタあたりのバイト数       ---  1024 バイト
ディスク容量                  ---  1250304 バイト
残り容量                      ---  508928 バイト

R>getd h: ⏎…ドライブHの表示⑤

*** ディスク情報表示プログラム Ver.1.1 ***

ドライブ H のディスク情報

FAT ID                        ---  FE                  ←(20Mバイトのハード・ディスク⑥)
1ドライブあたりのクラスタ数   ---  2468 クラスタ
1クラスタあたりのセクタ数     ---  8 セクタ
1セクタあたりのバイト数       ---  1024 バイト
ディスク容量                  ---  20217856 バイト
残り容量                      ---  6307840 バイト

R>
```

ドライブのディスク情報を，ファンクション 36H を用いて読み出し，引数 arg2 で指定された構造体中の各メンバにその得られたディスク情報を格納します．

サブルーチン func_1c は，引数 arg1 で指定されたドライブの FAT-ID アドレス(FAT 領域のディスク・バッファ)をファンクション 1CH によって読み出し，そのアドレスを far ポインタとして DX：AX レジスタに返します．

◆ **生成方法**

プログラム getd.exe は，次の手順で分割アセンブル/コンパイルして作成します．

masm /ML getdsub;

cl −J −As getd.c getdsub

◆ **実行サンプル**

リスト7-37 はディスク情報表示プログラム getd.exe の実行例を示しています．

① プログラム getd.exe をパラメータなしで起動する．

② ここで，カレント・ドライブは R ドライブ(RAM ディスク)になっているので，1.5 M バイト RAM ディスクのディスク情報が表示される．

③ プログラム getd.exe にドライブ名を与えて起動する．ここで，ドライブ Y はアサイン(ASSIGN)されたドライブであり，1 M バイトのフロッピ・ディスクである．

④ 1 M バイト・フロッピ・ディスクのディスク情報が表示される．

⑤ 次に，プログラム getd.exe にドライブ H を指定して起動する．ここで，ドライブ H は ASSIGN されたハード・ディスクである．

⑥ 20 M バイト・ハード・ディスクのディスク情報が表示される．

FCBによるファイル・アクセスを行う

リスト7-38(fcb.c)および**リスト7-39**(fcbsub.asm)は，FCBを用いてファイルのコピーを行うプログラムfcb.exeのソース・リストです．

● fcb.c

リスト7-38はプログラムfcb.exeのCソース部分です．同リストにおいて，構造体_DTAはファンクション4EHやファンクション4FHによってファイルの検索を行った結果，そのファイルに関する詳細な情報が格納されるデータ領域の構造を定義しています．構造体_FCBは，ファイルの読み書きを行う際に使用するFCBのデータ構造を定義しています．

関数mainでは最初にプログラム・タイトルを表示したのち，コマンド・ライン・パラメータの解析を行います．パラメータの解析では，デスティネーション・ファイル名が指定されていない場合は，そのポインタを文字列*.*へのポインタとしてセットします．

パラメータの解析が終わったら，関数copy_fcbにソース・ファイル名へのポインタおよびデスティネーション・ファイル名へのポインタを渡して，ファイルのコピー作業を行います．

関数copy_fcbは，FCBを用いたファイルのコピーを実現します．まず，最初に関数(サブルーチン)func_1aを用いてDTAの設定を行います．

次に，関数func_29を用いてソース・ファイル名の解析を行います．ここで，もしソース・ファイル名の指定でドライブの指定が間違っていれば，関数func_29から0FFHが返されるので，その旨をエラー表示してエラー・ストップします．

そして，関数func_11を用いて最初のファイルの検索を行います．これらの準備が終わったらdo～whileループに入ります．do～whileループでは，DTAに設定されている（関数func_11で検索された）FCB（ファイル情報）をライブラリ関数memcpyを用いて，それぞれソース側，デスティネーション側のFCBにコピーします．次に関数func_29を用いてデスティネーション・ファイル名の解析を行い，その結果をデスティネーションFCBに対して設定します．

これらの処理が終わったら，関数copy_nameを用いてファイル名の整合をとります．ファイル名の整合とは，ソース・ファイル名やデスティネーション・ファイル名でワイルド・カード(*や?)が使用されている場合に，ファイル名のワイルド・カード部分に，実際に検索されたファイル名の一部または全部の文字を当てはめていく作業をいいます．

これらのファイル名の整合が終わったら，ファイル名表示に使うためのファイル名をバッファに設定しておきます．

そして，ライブラリ関数memcmpによって，ソース側のドライブ名を含むファイル名がデスティネーション側で指定されたファイル名と同じかどうかを調べます．もし，ソース側とデスティネーション側のファイル名が同じであれば，コピーできないのでコピー作業は行いません．もし，ソース側とデスティネーション側のファイル名が違っている場合は，まず関数func_0fによってソース側のファイルをオープンします．次に，関数func_16によってデスティネーション側のファイルを新たに作成します．これによって，既存のデスティネーション・ファイルは更新されることになります．

これらのファイル・オープンやファイル作成に失敗したら，それぞれのエラー・メッセージを表示してプログラムfcb.exeをエラー・ストップします．ファイルのオープンや作成に成功したら，ソース・ファイルのサイズを読み出しておき，forループで無限ループに入ります．

forループでは，まず関数func_14を用いてソース・ファイルの読み出しを行います．ここで，関数func_14から返されるステータスが"1"であれば，ファイルのEOFに達したことになるのでforループから脱出します．もし，ステータスが"2"であればエラーが発生したことになるので，エラー表示したのちプログラムfcb.exeをエラー・ストップします．

このソース・ファイルの読み込みに成功したら，そのデータを関数func_15を用いてデスティネーション側のファイルに書き込みます．ここでも，関数func_15でエラーが発生した場合は，エラー表示してプログラムfcb.exeをエラー・ストップします．

関数func_14からのEOF検出によってforループを脱出したら，ソース側のFCBからファイルのサイズや最終更新日時をデスティネーション側のFCBにコピーします．これによって，ファイル更新日時などが保存されることになります．

そして，関数func_10を用いてソース側とデスティネーション側のファイルをそれぞれクローズします．ここで，もしファイルのクローズに失敗したら，その旨を表示してエラー・ストップします．ファイル・コピー作業に成功したら，そのソース・ファイル名とデスティネーション・ファイル名，およびファイル・サイズの表示を行います．このdo～whileループは，関数func_12によって次のファイルが検索され，該当するファイルがなくなるまで繰り返されます．

関数err_exitは，関数copy_fcb内でエラーが発生した場合に，ソース・ファイル，デスティネーショ

〔リスト7-38〕プログラム fcb.c ①

```
 1: /*************************************************************
 2: *
 3: *    機      能 :  ファイル・コピー（ＦＣＢによる）
 4: *    サ      ブ :  fcbsub.asm
 5: *    生      成 :  masm /ML fcbsub;
 6: *                  cl -J -AS fcb.c fcbsub
 7: *    使用方法 :  fcb [<ソース>] [<デスティネーション>]
 8: *
 9: **************************************************************/
10: #include      <stdio.h>
11: #include      <memory.h>
12: #define       S_MOD        0x0
13: #define       D_MOD        0x0
14: #define       FCB_SIZE     0x24
15: #define       NAME_SIZE    0x8
16: #define       EXT_SIZE     0x3
17: #define       REC_SIZE     0x8000
18:
19: /**************************************************************
20: *
21: *     構造体 :   _DTA
22: *     機  能 :   ＤＴＡのデータ構造の定義
23: *
24: ***************************************************************/
25: typedef struct _DTA {
26:     char atr0;                 /*00H         検索するファイルの属性 */
27:     char drv;                  /*01H         ドライブ番号(00H=A, 01H=B) */
28:     char file [NAME_SIZE];     /*02H - 09H パス名のファイル名部分 */
29:     char ext [EXT_SIZE];       /*0AH - 0CH パス名の拡張子部分 */
30:     char reserve [8];          /*0DH - 14H システム予約 */
31:     char atr;                  /*15H         検索されたファイルの属性 */
32:     int  time;                 /*16H - 17H 最終変更時刻 */
33:     int  date;                 /*18H - 19H 最終変更日付 */
34:     long size;                 /*1AH - 1DH ファイルの大きさ */
35:     char name [13];            /*1EH - 2AH パックされたファイル名 */
36:     char buff [1024];
37: } DTA;
38:
39: /**************************************************************
40: *
41: *     構造体 :   _FCB
42: *     機  能 :   FCBのデータ構造の定義
43: *
44: ***************************************************************/
45: typedef struct _FCB {
46:     char drv;                  /*00H         ドライブ番号 */
47:     char name [NAME_SIZE];     /*01H - 08H ファイル名 */
48:     char ext [EXT_SIZE];       /*09H - 0BH ファイル拡張子 */
49:     int  block;                /*0CH - 0DH カレント・ブロック */
50:     int  rec_size;             /*0EH - 0FH レコード・サイズ */
51:     long size;                 /*10H - 13H ファイル・サイズ */
52:     int  date;                 /*14H - 15H 最終更新日付 */
53:     int  time;                 /*16H - 17H 最終変更時刻 */
54:     char reserved [8];         /*18H - 1FH 予約領域 */
55:     char record;               /*20H         カレント・レコード */
56:     long rel_rec;              /*21H - 24H 相対レコード */
57: } FCB;
58:
59: void main (int, char **);
60: void copy_fcb (char *, char *);
61: void err_exit (FCB *, FCB *);
62: void copy_name (FCB *, FCB *);
63: int  func_0d (void);
64: int  func_0f (FCB *);
65: int  func_10 (FCB *);
66: int  func_11 (FCB *);
67: int  func_12 (FCB *);
68: int  func_14 (FCB *);
69: int  func_15 (FCB *);
70: void func_1a (DTA *);
71: int  func_29 (char *, FCB *, int);
72: char dta [REC_SIZE];
```

〔リスト7-38〕プログラム fcb.c ②

```
 73: /************************************************************
 74: *
 75: *    関数名:   main (argc, argv)
 76: *    機  能:   パラメータ（オプション）の解析
 77: *    入  力:   int argc        ·····  コマンドライン・パラメータの数
 78: *              char *argv[]    ·····  パラメータ文字列へのポインタ
 79: *    出  力:   なし
 80: *
 81: ************************************************************/
 82: void main (argc, argv)
 83: int argc;
 84: char **argv;
 85: {
 86:     char *s_ptr, *d_ptr;
 87:
 88:     printf ("¥n *** ファイル・コピー（ＦＣＢ）プログラム Ver.1.1 ***¥n¥n");
 89:     if (argc == 1) {
 90:         printf ("使用法: fcb ソース名 デスティネーション名¥n");
 91:         exit (1);
 92:     }
 93:     s_ptr = *++argv;
 94:     d_ptr = *++argv;
 95:     if (argc == 2) {
 96:         d_ptr = "*.*";
 97:     }
 98:     copy_fcb (s_ptr, d_ptr);
 99:     exit (0);
100: }
101:
102: /************************************************************
103: *
104: *    関数名:   copy_fcb (s_ptr, d_ptr)
105: *    機  能:   ファイルのコピー
106: *    入  力:   s_ptr    ·····  ソース・ファイル名へのポインタ
107: *              d_ptr    ·····  デスティネーション・ファイル名へのポインタ
108: *    出  力:   なし
109: *
110: ************************************************************/
111: void copy_fcb (s_ptr, d_ptr)
112: char *s_ptr, *d_ptr;
113: {
114:     FCB fcb_b, fcb_s, fcb_d;
115:     char s_name [NAME_SIZE + EXT_SIZE + 2];
116:     char d_name [NAME_SIZE + EXT_SIZE + 2];
117:     int st;
118:     long bytes;
119:
120:     func_1a ((DTA *)dta);                        /* DTA セット */
121:     if (func_29 (s_ptr, &fcb_b, S_MOD) == 0xFF) { /* ファイル名の解析 */
122:         printf ("ドライブの指定が無効です.¥n");
123:         exit (1);
124:     }
125:     if (func_11 (&fcb_b)) {                      /* 最初のファイルの検索 */
126:         printf ("ファイルが存在しません.¥n");
127:         exit (2);
128:     }
129:     do {
130:         memcpy ((char *) &fcb_s, dta, FCB_SIZE);
131:         memcpy ((char *) &fcb_d, dta, FCB_SIZE);
132:         if (func_29 (d_ptr, &fcb_d, D_MOD) == 0xFF) { /* ファイル名の解析 */
133:             printf ("ドライブの指定が無効です.¥n");
134:             exit (3);
135:         }
136:         copy_name (&fcb_d, &fcb_s);              /* ファイル名セット */
137:         memcpy (s_name, fcb_s.name, NAME_SIZE);
138:         s_name [NAME_SIZE] = '.';
139:         memcpy (s_name + NAME_SIZE + 1, fcb_s.ext, EXT_SIZE);
140:         s_name [NAME_SIZE + EXT_SIZE + 1] = '¥0';
141:
142:         memcpy (d_name, fcb_d.name, NAME_SIZE);
143:         d_name [NAME_SIZE] = '.';
144:         memcpy (d_name + NAME_SIZE + 1, fcb_d.ext, EXT_SIZE);
145:         d_name [NAME_SIZE + EXT_SIZE + 1] = '¥0';
146:
147:         if (memcmp ((char *) &fcb_s, (char *) &fcb_d,
148:                     NAME_SIZE + EXT_SIZE + 2)) {
149:             if (func_0f (&fcb_s)) {              /* ソース・オープン */
150:                 printf ("ファイル '%s' がありません.¥n", s_name);
```

〔リスト7-38〕プログラム fcb.c ③

```
151:            exit (4);
152:        }
153:        if (func_16 (&fcb_d)) {
154:            printf ("ファイル '%s' が作れません.¥n", d_name);
155:            err_exit (&fcb_d, &fcb_s);
156:        }
157:        bytes = fcb_s.size;
158:        for (; ;) {
159:            st = func_14 (&fcb_s);                  /* 読み込み */
160:            if (st == 1) {
161:                break;
162:            }
163:            if (st == 2) {
164:                printf ("ファイル '%s' の読み込みができません.¥n", s_name);
165:                err_exit (&fcb_d, &fcb_s);
166:            }
167:            if (func_15 (&fcb_d)) {                 /* 書き込み */
168:                printf ("ファイル '%s' への書き込みができません.¥n", d_name);
169:                err_exit (&fcb_d, &fcb_s);
170:            }
171:        }
172:        if (func_10 (&fcb_s)) {
173:            printf ("ファイル '%s' がクローズできません.¥n", s_name);
174:            exit (5);
175:        }
176:        fcb_d.size = fcb_s.size;                    /* サイズのコピー */
177:        fcb_d.time = fcb_s.time;                    /* 時刻のコピー */
178:        fcb_d.date = fcb_s.date;                    /* 日付のコピー */
179:        if (func_10 (&fcb_d)) {
180:            printf ("ファイル '%s' がクローズできません.¥n", d_name);
181:            exit (6);
182:        }
183:        printf ("%c:%s を %c:%s にコピーしました (%ld バイト) ¥n",
184:                *(char *)&fcb_s + '@', s_name,
185:                *(char *)&fcb_d + '@', d_name, fcb_s.size);
186:    }
187: } while (! func_12 (&fcb_b));            /* つぎのファイル検索 */
188: }
189:
190: /*****************************************************************
191: *
192: *    関数名:   err_exit (d_ptr, s_ptr)
193: *    機  能:   エラー終了
194: *    入  力:   d_ptr  ･･･ デスティネーションFCBへのポインタ
195: *              s_ptr  ･･･ ソースFCBへのポインタ
196: *    出  力:   なし
197: *
198: *****************************************************************/
199: void err_exit (d_ptr, s_ptr)
200: FCB *d_ptr, *s_ptr;
201: {
202:     func_10 (d_ptr);                  /* ソースクローズ */
203:     func_10 (s_ptr);                  /* デスティネーション・クローズ */
204:     exit (2);
205: }
206:
207: /*****************************************************************
208: *
209: *    関数名:   copy_name (d_ptr, s_ptr)
210: *    機  能:   ファイル名のコピー
211: *    入  力:   d_ptr  ･･･ デスティネーション側FCBへのポインタ
212: *              s_ptr  ･･･ ソース側FCBへのポインタ
213: *    出  力:   なし
214: *
215: *****************************************************************/
216: void copy_name (d_ptr, s_ptr)
217: FCB *d_ptr, *s_ptr;
218: {
219:     char *s, *d;
220:     int i;
221:
222:     s = s_ptr -> name;
223:     d = d_ptr -> name;
224:     for (i = 0; i < NAME_SIZE + EXT_SIZE; i++) {
225:         if (*d == '?' && *s != '?') {
226:             *d++ = *s++;
227:         }
228:     }
229: }
```

ン・ファイルをそれぞれクローズしたのち，終了コード4をもってプログラム fcb.exe を終了します．

関数 copy_name は，ソース・ファイル名とデスティネーション・ファイル名へのポインタを引数として受け取り，ワイルド・カード(*や?)で展開されているFCB内のデスティネーション・ファイル名に対して，ソース・ファイル名の該当する位置の文字をオーバライトしていきます．

● fcbsub.asm

リスト7-39 はプログラム fcb.exe のアセンブリ・ソース部分です．同リストにおいて，サブルーチン(関数) func_0f は，arg1 で指定された FCB とファンクション 0FH を用いてファイルのオープンを行います．ここで，もしファイルのオープンに成功したら AX レジスタには0を返し，ファイル・オープンに失敗したら AX レジスタに 0FFH を返します．

サブルーチン func_10 は，arg1 で指定された FCB とファンクション 10H を用いてオープンされているファイルのクローズを行います．ここでも，もしファイルのクローズに成功したら AX レジスタには0を返し，ファイル・クローズに失敗したら AX レジスタに 0FFH を返します．

サブルーチン func_11 は，arg1 の FCB で指定されたワイルド・カードを含むファイル名に対して，ファンクション 11H を用いて最初のエントリ(ファイル

〔リスト7-39〕
プログラム
fcbsub.asm ①

```
 1: ;**********************************************************************
 2: ;
 3: ;   機      能 :   ＦＣＢによるファイル・アクセス・サブルーチン
 4: ;   ファンクション :   0FH（ファイルのオープン）
 5: ;                  10H（ファイルのクローズ）
 6: ;                  11H（最初のディレクトリ・エントリの検索）
 7: ;                  12H（つぎのディレクトリ・エントリの検索）
 8: ;                  14H（シーケンシャルな読み出し）
 9: ;                  15H（シーケンシャルな書き込み）
10: ;                  1AH（ＤＴＡアドレスの設定）
11: ;                  29H（ファイル名の解析）
12: ;   生      成 :   masm /ML fcbsub;
13: ;
14: ;**********************************************************************
15:             .MODEL   SMALL, C
16:             .CODE
17: ;**********************************************************************
18: ;
19: ;    ルーチン名 :   func_0f
20: ;    機    能 :   ファイルのオープン
21: ;    func     :   1FH（ＦＣＢによるファイルのオープン）
22: ;    入    力 :   arg1  ･･･ オープンされていないＦＣＢへのポインタ
23: ;    出    力 :   AX    ･･･ エラーなし :   0
24: ;                          エラーあり :   FFh
25: ;
26: ;**********************************************************************
27: func_0f     PROC      arg1:PTR
28:             mov       dx, arg1
29:             mov       ah, 0Fh                        ;ファンクション 0FH
30:             int       21h
31:             xor       ah, ah
32:             ret
33: func_0f     ENDP
34:
35: ;**********************************************************************
36: ;
37: ;    ルーチン名 :   func_10
38: ;    機    能 :   ファイルのクローズ
39: ;    func     :   10H（ＦＣＢによるファイルのクローズ）
40: ;    入    力 :   arg1  ･･･ オープンされているＦＣＢへのポインタ
41: ;    出    力 :   AX    ･･･ エラーなし :   0
42: ;                          エラーあり :   FFh
43: ;
44: ;**********************************************************************
45: func_10     PROC      arg1:PTR
46:             mov       dx, arg1
47:             mov       ah, 10h                        ;ファンクション 10H
48:             int       21h
49:             xor       ah, ah
50:             ret
51: func_10     ENDP
52:
```

名)の検索を行います．この関数でも，ファイルが存在する場合にはAXレジスタに0を返し，ファイルが存在しない場合にはAXレジスタに0FFHを返します．

サブルーチンfunc_12は，arg1のFCBで指定されたファイル名に対して，ファンクション12Hを用いて次のファイルの検索を行います．ここでも，ファイルが存在する場合にはAXレジスタに0を返し，ファイルが存在しない場合にはAXレジスタに0FFHを返します．

サブルーチンfunc_14は，arg1で指定されたFCB(ファイル)に対して，ファンクション14Hを用いてファイルのシーケンシャルな読み出しを行います．ここで，ファイルの読み出しに成功したらAXレジスタに0を返し，もしファイルの読み出しに失敗したらAXレジスタにそのエラー・コードを返します．

サブルーチンfunc_15は，arg1で指定されたFCB(ファイル)に対して，ファンクション15Hを用いてデータのシーケンシャルな書き込みを行います．ここで，ファイルの書き込みに成功したらAXレジスタに0を返し，もしファイルの書き込みに失敗したらAXレジスタにそのエラー・コードを返します．

サブルーチンfunc_16は，ファンクション16Hを用いて，arg1(FCB)で指定されたファイルを新たに作成します．このとき，もし指定されたファイルがすでに存在する場合は，そのファイルが更新されることになります．この関数でも，ファンクション16Hにおい

〔リスト7-39〕
プログラム
fcbsub.asm ②

```
 53: ;**********************************************************************
 54: ;
 55: ;      ルーチン名 :   func_11
 56: ;      機　能 :       最初のエントリの検索
 57: ;      func   :       11H（ＦＣＢによる最初のエントリの検索）
 58: ;      入　力 :       arg1   ･･･ オープンされていないＦＣＢへのポインタ
 59: ;      出　力 :       AX     ･･･ エラーなし :  0
 60: ;                                エラーあり :  FFh
 61: ;
 62: ;**********************************************************************
 63: func_11         PROC    arg1:PTR
 64:                 mov     dx, arg1
 65:                 mov     ah, 11h                 ;ファンクション11H
 66:                 int     21h
 67:                 xor     ah, ah
 68:                 ret
 69: func_11         ENDP
 70:
 71: ;**********************************************************************
 72: ;
 73: ;      ルーチン名 :   func_12
 74: ;      機　能 :       つぎのエントリの検索
 75: ;      func   :       12H（ＦＣＢによる最初のエントリの検索）
 76: ;      入　力 :       arg1   ･･･ オープンされていないＦＣＢへのポインタ
 77: ;      出　力 :       AX     ･･･ エラーなし :  0
 78: ;                                エラーあり :  FFh
 79: ;
 80: ;**********************************************************************
 81: func_12         PROC    arg1:PTR
 82:                 mov     dx, arg1
 83:                 mov     ah, 12h                 ;ファンクション12H
 84:                 int     21h
 85:                 xor     ah, ah
 86:                 ret
 87: func_12         ENDP
 88:
 89: ;**********************************************************************
 90: ;
 91: ;      ルーチン名 :   func_14
 92: ;      機　能 :       データの読み出し
 93: ;      func   :       14H（ＦＣＢによるシーケンシャルなファイルの読み出し）
 94: ;      入　力 :       arg1   ･･･ オープンされているＦＣＢへのポインタ
 95: ;      出　力 :       AX     ･･･ エラーなし :  0
 96: ;                                エラーあり :  エラー・コード
 97: ;
 98: ;**********************************************************************
 99: func_14         PROC    arg1:PTR
100:                 mov     dx, arg1
101:                 mov     ah, 14h                 ;ファンクション14H
102:                 int     21h
103:                 xor     ah, ah
104:                 ret
105: func_14         ENDP
```

〔リスト7-39〕
プログラム
fcbsub.asm ③

```
106:
107: ;*********************************************************************
108: ;
109: ;       ルーチン名 :   func_15
110: ;       機   能 :   データの書き込み
111: ;       func    :   15H（ＦＣＢによるシーケンシャルなファイルの書き込み）
112: ;       入   力 :   arg1   ･･･ オープンされているＦＣＢへのポインタ
113: ;       出   力 :   AX     ･･･ エラーなし :   0
114: ;                              エラーあり :   エラー・コード
115: ;
116: ;*********************************************************************
117: func_15         PROC    arg1:PTR
118:                 mov     dx, arg1
119:                 mov     ah, 15h                 ;ファンクション 15H
120:                 int     21h
121:                 xor     ah, ah
122:                 ret
123: func_15         ENDP
124:
125: ;*********************************************************************
126: ;
127: ;       ルーチン名 :   func_16
128: ;       機   能 :   ファイルの作成
129: ;       func    :   16H（ＦＣＢによるファイルの作成）
130: ;       入   力 :   arg1   ･･･ オープンされていないＦＣＢへのポインタ
131: ;       出   力 :   AX     ･･･ エラーなし :   0
132: ;                              エラーあり :   FFh
133: ;
134: ;*********************************************************************
135: func_16         PROC    arg1:PTR
136:                 mov     dx, arg1
137:                 mov     ah, 16h                 ;ファンクション 16H
138:                 int     21h
139:                 xor     ah, ah
140:                 ret
141: func_16         ENDP
142:
143: ;*********************************************************************
144: ;
145: ;       ルーチン名 :   func_1a
146: ;       機   能 :   ＤＴＡアドレス設定
147: ;       func    :   1AH（ディスク転送アドレスの設定）
148: ;       入   力 :   arg1   ･･･ DTAへのポインタ
149: ;       出   力 :   なし
150: ;
151: ;*********************************************************************
152: func_1a         PROC    arg1:PTR
153:                 mov     dx, arg1
154:                 mov     ah, 1Ah                 ;ファンクション 1AH
155:                 int     21h
156:                 xor     ah, ah
157:                 ret
158: func_1a         ENDP
159:
160: ;*********************************************************************
161: ;
162: ;       ルーチン名 :   func_29
163: ;       機   能 :   ファイル名の解析
164: ;       func    :   29H（ＦＣＢによるファイル名の解析）
165: ;       入   力 :   arg1   ･･･ 解析する文字列へのポインタ
166: ;                   arg2   ･･･ オープンされていないＦＣＢへのポインタ
167: ;                   arg3   ･･･ 解析制御ワード
168: ;       出   力 :   AX     ･･･ エラーなし :   0 （ワイルド・カードなし）
169: ;                              エラーなし :   1 （ワイルド・カードあり）
170: ;                              エラーあり :   FFh
171: ;
172: ;*********************************************************************
173: func_29         PROC    arg1:PTR, arg2:PTR, arg3:WORD
174:                 push    di
175:                 push    si
176:                 mov     si, arg1
177:                 mov     di, arg2
178:                 mov     ax, arg3
179:                 mov     ah, 29h                 ;ファンクション 29H
180:                 int     21h
181:                 xor     ah, ah
182:                 pop     si
183:                 pop     di
184:                 ret
185: func_29         ENDP
186:                 END
```

て，ファイルの作成に成功したらAXレジスタに0を返し，もしファイルの作成に失敗したらAXレジスタには0FFHを返します．

サブルーチンfunc_1aは，ファンクション1aHを用いてarg1で指定されたDTAを設定します．

サブルーチンfunc_29は，ファンクション29Hを用いてarg1で指定されたワイルド・カードを含むファイル名の解析を行います．ここで，解析された結果はarg2で指定されたFCBの中に返されます．このとき，解析モードは，arg3によって指定されたビットにより**表6-14**(169ページ)にしたがって制御されます．ファイル名の解析の結果，ワイルド・カードが使用されていなければAXレジスタに0を返し，ワイルド・カードが使用されている場合は1を返します．また，ファイル名の解析でエラーが発生した場合には，AXレジスタに0FFHを返します．

◆ **生成方法**

プログラムfcb.exeは，下記の手順で分割アセンブル/コンパイルして作成します．

```
masm /ML fcbsub;
cl -J -As fcb.c fcbsub
```

◆ **実行サンプル**

リスト7-40はFCBによるファイル・コピーのテスト・プログラムfcb.exeの実行例を示しています．

① dirコマンドを用いてカレント・ドライブ上の拡張子.bakのファイルを確認する．

② ここで，拡張子.bakのファイルが存在する．これらのファイルは，このあとに更新されるのでファイル・サイズや最終更新日時に注目．

③ 次に，dirコマンドを用いてドライブH上の拡張子.cのファイルを確認する．

④ ここで拡張子.cのファイルが存在する．これらのファイルは，このあとにソース側のファイルとなるのでファイル・サイズや更新日時に注目．

⑤ プログラムfcb.exeを用いてドライブH上の拡張子.cをもつファイルをカレント・ドライブ上の拡張子.bakのファイルにコピーし，ソース・ファイルのバックアップを行う．

⑥ プログラムfcb.exeからソース・ファイル名，デスティネーション・ファイル名，ファイル・サイズなどの表示が行われる．

⑦ dirコマンドを用いてコピーされた拡張子.bakのファイルを確認する．

⑧ 指定どおりに拡張子.bakのファイルが書き替えられて更新されている．

⑨ fcコマンドを用いてファイルの比較を行い，正確にバックアップされたかを確認する．

⑩ 正確にコピーされている．

● PC-9801専用エスケープ・シーケンス ●

PC-9801では，MS-DOS標準のエスケープ・シーケンスのほかに，つぎのエスケープ・シーケンスが用意されていて，キー割り当ての変更を行うことが可能になっています．

```
ESC [Pn ; … ; Pnp
  または
ESC ["string" ; Pnp
  または
ESC [Pn ; "string" ; Pnp
```

ここでPnにはキー・コードを10進数で与えます．stringの長さは15文字まででです．

【例】

① ESC [62 ; 82p

"R"キーに"B"を割り当てるので，"R"キーを押すと"B"が入力される．

② ESC ["Ddir" ; 13p

"D"キーに"dir"とキャリッジ・リターン・コードを割り当てるので，以後"D"キーを押すとdirコマンドが実行される．

〔リスト7-40〕プログラム fcb.exe の実行例

```
R>dir *.bak ⏎ … 拡張子 .bak のファイルを確認 ①

 ドライブ R: のディスクのボリュームラベルは [IOS-10    ]
 ディレクトリは R:¥WK

CTRL      BAK     2945  88-12-02  11:49 ←(拡張子 .bak の存在に注目 ②)
FATAL     BAK     1411  88-12-02  11:52
DDUMP     BAK     4916  88-12-01  17:11



GINT      BAK      369  88-12-08  14:08
STMP      BAK     2748  88-12-09  11:33
FCB       BAK     2164  88-12-13  17:06
        25 個のファイルがあります.
    290816 バイトが使用可能です.

R>dir h:*.c ⏎ … 拡張子 .c のファイルを確認 ③

 ドライブ H: のディスクのボリュームラベルは HD1
 ディレクトリは H:¥WK1¥WK

CTRL      C       1366  88-12-12   9:16 ←(拡張子 .c の存在に注目 ④)
FATAL     C       3620  88-12-12  10:07
DDUMP     C       6058  88-12-12  10:50



GINT      C       1828  88-12-12  16:25
STMP      C       5766  88-12-12  16:32
FCB       C       7476  88-12-21  17:03
        34 個のファイルがあります.
   5128192 バイトが使用可能です.

R>fcb h:*.c *.bak ⏎ … プログラム fcb.exe を用いてドライブ H の .c をカレント・ドライブの .bak にコピーする ⑤

 *** ファイル・コピー (FCB) プログラム Ver.1.1 ***

H:CTRL    .C  を  R:CTRL    .BAK にコピーしました (1366 バイト)  ┐
H:FATAL   .C  を  R:FATAL   .BAK にコピーしました (3620 バイト)  │
H:DDUMP   .C  を  R:DDUMP   .BAK にコピーしました (6058 バイト)  │
                 .                                              │
                 .                                              ├ fcb.exe からのメッセージ ⑥
                 .                                              │
H:GINT    .C  を  R:GINT    .BAK にコピーしました (1828 バイト)  │
H:STMP    .C  を  R:STMP    .BAK にコピーしました (5766 バイト)  │
H:FCB     .C  を  R:FCB     .BAK にコピーしました (7476 バイト)  ┘

R>dir *.bak ⏎ … 拡張子 .bak のファイルを確認 ⑦
 ドライブ R: のディスクのボリュームラベルは [IOS-10    ]
 ディレクトリは R:¥WK

CTRL      BAK     1366  88-12-12   9:16 ←(指定どおりに .bak ファイルが書き換えられている ⑧)
FATAL     BAK     3620  88-12-12  10:07
DDUMP     BAK     6058  88-12-12  10:50



GINT      BAK     1828  88-12-12  16:25
STMP      BAK     5766  88-12-12  16:32
FCB       BAK     7476  88-12-21  17:03
        37 個のファイルがあります.
    225280 バイトが使用可能です.

R>fc h:fcb.c fcb.bak /b ⏎ … MS-DOS の FC コマンドを用いてファイル内容を照合 ⑨
FC: 違いは見つかりません. ←(正確にコピーされている ⑩)

R>
```

ファイルのタイム・スタンプを変更する

リスト7-41(stmp.c)および**リスト7-42**(stmpsub.asm)は，指定されたファイル(ワイルド・カードも含む)のタイム・スタンプ(最終更新日時)の変更を行うプログラムstmp.exeのソース・リストです．

● stmp.c

リスト7-41はプログラムstmp.exeのCソース部分です．同リストにおいて，構造体DTAは，ファンクション4EHやファンクション4FHにおいて，検索されたファイルに関する情報の格納されるデータ領域の構造を定義します．

構造体_STMPは，ファンクション59Hを用いて指定したファイルのタイム・スタンプを読み出した際に，その得られたタイム・スタンプを格納するためのデータ領域の構造を定義します．

関数mainでは，最初にプログラム・タイトルを表示したのち，コマンド・ライン・パラメータの解析を行います．パラメータの解析ではオプションが指定されているかどうかを調べます．もし-dオプションが指定されている場合には，関数set_dateを用いてコマンド・ラインで指定された日付を変数dateに格納します．同様に，-tオプションの場合には，関数set_timeを用いてコマンド・ラインで指定された時刻を変数timeに格納しておきます．

コマンド・ライン・パラメータがオプションでない場合は，その文字列をファイル名と解釈して文字列ポインタ(src_file)の設定を行います．ここで，もし指定されたファイル名がパス名の場合には，そのディレクトリ名の部分をバッファpathに切り出し，ファイル名と分離しておきます．

これらの処理が終わったら，関数stmp_setを用いて指定したディレクトリにあるファイルのタイム・スタンプの変更を行います．

〔リスト7-41〕
プログラム
stmp.c ①

```
 1: /************************************************************************
 2: *
 3: *   機    能:   タイム・スタンプの変更
 4: *   サ    ブ:   stmpsub.asm
 5: *   生    成:   masm /ML stmpsub;
 6: *               cl -J -AS stmp.c stmpsub
 7: *   使用方法:   stmp <ファイル名> [<-d日付>] [<-t時刻>]
 8: *
 9: ************************************************************************/
10: #include     <stdio.h>
11: #include     <ctype.h>
12: #include     <stdlib.h>
13: #include     <string.h>
14: #define      ATR_READ      0xFFFF
15: #define      READ_MOD      0
16:
17: /************************************************************************
18: *
19: *    構造体:   _DTA
20: *    機  能:   DTAのデータ構造の定義
21: *
22: ************************************************************************/
23: typedef struct _DTA {
24:      char atr0;                  /*00H      検索するファイルの属性 */
25:      char drv;                   /*01H      ドライブ番号(00H=A, 01H=B) */
26:      char file [8];              /*02H - 09H パス名のファイル名部分 */
27:      char ext [3];               /*0AH - 0CH パス名の拡張子部分 */
28:      char reserve [8];           /*0DH - 14H システム予約 */
29:      char atr;                   /*15H      検索されたファイルの属性 */
30:      int  time;                  /*16H - 17H 最終変更時刻 */
31:      int  date;                  /*18H - 19H 最終変更日付 */
32:      long size;                  /*1AH - 1DH ファイルの大きさ */
33:      char name [13];             /*1EH - 2AH パックされたファイル名 */
34: } DTA;
35:
36: /************************************************************************
37: *
38: *    構造体:   _STMP
39: *    機  能:   タイム・スタンプ・バッファのデータ構造の定義
40: *
41: ************************************************************************/
42: typedef struct _STMP {
43:      int time;
44:      int date;
45: } STMP;
46:
```

〔リスト7-41〕
プログラム
stmp.c ②

```
 47: void main (int, char **);
 48: int  set_date (char *);
 49: int  set_time (char *);
 50: void stmp_set (char *, char *, int, int);
 51: void stmp_read (int, STMP *);
 52: void stmp_write (int, STMP *);
 53: void func_1a (DTA *);                   /* ＤＴＡの設定 */
 54: int  func_3d (char *, int);             /* ファイルのオープン */
 55: int  func_3e (int);                     /* ファイルのクローズ */
 56: int  func_4e (char *, int);             /* 一致するファイルの検索 */
 57: int  func_4f (void);                    /* つぎに一致するファイルの検索 */
 58: /*******************************************************************************
 59: *
 60: *    関数名：   main (argc, argv)
 61: *    機  能：   パラメータ（オプション）の解析
 62: *    入  力：   int argc      ・・・・ コマンドライン・パラメータの数
 63: *               char *argv[]  ・・・ パラメータ文字列へのポインタ
 64: *    出  力：   なし
 65: *
 66: *******************************************************************************/
 67: void main (argc, argv)
 68: int argc;
 69: char **argv;
 70: {
 71:     char *src_file, *path_ptr;
 72:     char path [256];
 73:     int date, time;
 74:
 75:     printf ("¥n *** タイム・スタンプ変更プログラム Ver.1.1 ***¥n");
 76:     date = time = 0;
 77:     while (--argc > 0) {
 78:         if (**++argv == '-') {
 79:             if (isupper (argv[0][1])) {
 80:                 argv[0][1] = tolower (argv[0][1]);
 81:             }
 82:             switch (argv[0][1]) {
 83:             case 'd' :
 84:                 date = set_date (*argv + 2);
 85:                 break;
 86:
 87:             case 't' :
 88:                 time = set_time (*argv + 2);
 89:                 break;
 90:
 91:             default:
 92:                 printf ("option error :¥"%c¥"¥n", argv[0][1]);
 93:                 exit (1);
 94:             }
 95:         } else {
 96:             src_file = *argv;
 97:         }
 98:     }
 99:     if (strchr (src_file, '¥¥') != NULL) {
100:         strcpy (path, src_file);
101:     }
102:     path_ptr = path;
103:     if ((path_ptr = strrchr (path_ptr, '¥¥')) != NULL) {
104:         *++path_ptr = '¥0';
105:     } else {
106:         *path = '¥0';
107:     }
108:     stmp_set (path, src_file, date, time);
109:     exit (0);
110: }
111:
112: /*******************************************************************************
113: *
114: *    関数名：   set_date (ptr)
115: *    機  能：   日付の設定
116: *    入  力：   ptr  ・・・ -dオプションへのポインタ
117: *    出  力：   date ・・・ 日付データ
118: *
119: *******************************************************************************/
120: int set_date (ptr)
121: char *ptr;
122: {
123:     int date, yy, mm, dd;
124:
125:     yy = mm = dd = 0;
126:     yy = atoi (ptr);
127:     if (yy > 1900) {
```

〔リスト7-41〕
プログラム
stmp.c ③

```
128:           yy -= 1900;
129:       }
130:       if (yy > 80) {
131:           yy -= 80;
132:       }
133:       if ((ptr = strchr (ptr, '-')) != NULL) {
134:           mm = atoi (++ptr);
135:           if ((ptr = strchr (ptr, '-')) != NULL) {
136:               dd = atoi (++ptr);
137:           }
138:       }
139:       date = (yy << 9) | (mm << 5) | dd;
140:       return (date);
141: }
142:
143: /*************************************************************************
144:  *
145:  *   関数名:   set_time (ptr)
146:  *   機  能:   日付の設定
147:  *   入  力:   ptr  ··· -dオプションへのポインタ
148:  *   出  力:   time ··· 日付データ
149:  *
150:  *************************************************************************/
151: int set_time (ptr)
152: char *ptr;
153: {
154:       int time, hh, mm, ss;
155:
156:       hh = mm = ss = 0;
157:       hh = atoi (ptr);
158:       if ((ptr = strchr (ptr, ':')) != NULL) {
159:           mm = atoi (++ptr);
160:           if ((ptr = strchr (ptr, ':')) != NULL) {
161:               ss = atoi (++ptr);
162:           }
163:       }
164:       time = (hh << 11) | (mm << 5) | ss;
165:       return (time);
166: }
167:
168: /*************************************************************************
169:  *
170:  *   関数名:   stmp_set (path, file, date, time)
171:  *   機  能:   タイム・スタンプの更新
172:  *   入  力:   path ··· パス名へのポインタ
173:  *             file ··· ファイル名へのポインタ(パスを含む)
174:  *             date ··· パックされた日付データ
175:  *             time ··· パックされた時刻データ
176:  *   出  力:   なし
177:  *
178:  *************************************************************************/
179: void stmp_set (path, file, date, time)
180: char *path, *file;
181: int date, time;
182: {
183:       char f_name [256];
184:       DTA dta;
185:       STMP stmp;
186:       int fp;
187:
188:       func_1a (&dta);                              /* DTAの設定 */
189:       if (func_4e (file, ATR_READ)) {             /* 一致するファイルの検索 */
190:           printf ("¥nファイルがありません.¥n¥n");
191:           exit (2);
192:       }
193:       do {
194:           strcpy (f_name, path);
195:           strcat (f_name, dta.name);
196:           fp = func_3d (f_name, READ_MOD);        /* ファイルのオープン */
197:           stmp_read (fp, &stmp);
198:           if (date) {
199:               stmp.date = date;
200:           }
201:           if (time) {
202:               stmp.time = time;
203:           }
204:           stmp_write (fp, &stmp);
205:           func_3e (fp);                            /* ファイルのクローズ */
206:       } while (! func_4f ());                      /* つぎのファイル検索 */
207: }
```

関数set_dateでは，コマンド・ラインの−dオプション文字列へのポインタを引数ptrとして受け取り，“−”に出会うたびに年，月，日の順に文字列を10進数値に変換します．ここで，年は1980がベースとなるので最初に1900を差し引き，次に80を差し引いて−dオプションの指定時に1900を省略できるようにしています．

これらの処理が終わったら，日付のフォーマット(第3章参照)に合わせるため日付の各データをビット・シフトし，それらの論理和(OR)をとって2バイトの変数dateに格納して戻ります．

関数set_timeでは，関数set_dateと同様にコマンド・ラインの−tオプション文字列へのポインタを引数ptrとして受け取り，“：”に出会うたびに時，分，秒の順に文字列を数値に変換します．数値への変換が終わったら，時刻のフォーマット(第3章参照)に合わせるため時刻の各データをビット・シフトし，それらのORをとって2バイトの変数timeに格納して戻ります．

関数stmp_setでは，関数(サブルーチン)func_1aによってDTAアドレスの設定を行い，次に関数func_4eを用いて，引数で指定されたファイル名(ワイルド・カードを含む)に一致するファイルの検索を行います．

一致するファイルが見つかったらdo〜whileループに入ります．do〜whileループでは，ファンクション4EHまたはファンクション4FHによって検索されたファイル名(パス名ではない)がDTAのメンバnameに返されるので，そのファイル名と指定されたディレクトリ名pathを連結してパス名とし，関数func_3dに渡して，そのファイルのオープンを行って変数fpにファイル・ポインタを格納します．

ファイル・ポインタの格納に成功したら，タイム・スタンプの設定を行います．ここで，コマンド・ラインの指定では，タイム・スタンプの日付と時刻の指定は独立して行うことができ，もしどちらかのタイム・スタンプの指定が省略されている場合は，その変数(dateまたはtime)の内容が0になっているので，その場合は読み出したタイム・スタンプを変更せず，オプションの指定によって日付や時刻のデータが入っているときだけ，タイム・スタンプのデータを変更します．

タイム・スタンプのデータ設定が終わったら，関数stmp_writeを用いてタイム・スタンプの変更を行い，関数func_3eを用いてファイルをクローズすることによってタイム・スタンプの更新が行われます．

これらの処理が終わったら，関数func_4fを用いて次に一致するファイルを検索し，指定されたファイルが存在しなくなるまでdo〜whileループを繰り返します．

● stmpsub.asm

リスト7-42はプログラムstmp.exeのアセンブリ・ソース部分です．同リストにおいて，ストラクチャ_STMPは，ファンクション57Hによって得られたタイム・スタンプを格納するためのデータ領域の構造を定義します．

サブルーチン(関数)func_1aは，ファンクション1AHを用いてarg1によって指定されたDTAアドレスの設定を行います．

サブルーチンfunc_3dは，ファンクション3DHを用いて，arg1で指定されたファイルをarg2で指定された読み書きのモードでオープンします．ファイルのオープンに成功すると，ファンクション3DHからAXレジスタにファイル・ハンドルが返されるので，そのレジスタ内容をそのまま返します．もし，ファイル・オープンに失敗すると，AXレジスタにエラー・コードが返されるので，そのAXレジスタのビット15をセットしてファイル・ハンドルと区別できるようにして返します．

サブルーチンfunc_3eでは，ファンクション3EHを用いてarg1で指定されたファイル・ハンドルのクローズを行います．ここで，もしファイルのクローズに失敗したら，そのエラー・コードをAXレジスタに返し，成功したらAXレジスタに0を返します．

サブルーチンfunc_4eは，ファンクション4EHを用いてarg1で指定されたファイル名(ワイルド・カードを含む)とarg2で指定された属性をもつファイルの検索を行います．もし，指定されたファイルが見つかれば，そのファイルに関する詳しい情報がDTAに返されるのでAXレジスタを0にして戻ります．もし，ファイルがないなどのエラーが発生した場合には，そのエラー・コードをAXレジスタに返します．

サブルーチンfunc_4fは，次に一致するファイルの検索を行います．ここでも，一致するファイルが見つかると，そのファイルの情報がDTAに返されるのでAXレジスタを0にして戻ります．もし，次のファイルが存在しない場合には，そのエラー・コードがAXレジスタに返されるのでそのまま戻ります．

サブルーチンstmp_readでは，ファンクション57Hを用いてarg1で指定されたファイルのタイム・スタンプを読み込んで，arg2で指定されたデータ領域に格納します．ここで，ファンクション57Hでエラーがなければ AXレジスタに0を返し，エラーが発生した場合には，そのエラー・コードをAXレジスタに返します．

サブルーチンstmp_writeでは，ファンクション57Hを用いて，arg1で指定されたファイルに対しarg2で指定された領域にあるタイム・スタンプを書き出します．ここでも，ファンクション57Hでエラーが発生すればAXレジスタにエラー・コードを返し，正常に終了した場合にはAXレジスタに0を返します．

◆ 生成方法

プログラムstmp.exeは，以下のような手順で分割アセンブル/コンパイルして作成します．

```
masm /ML stmpsub;
cl -J -As stmp.c stmpsub
```

〔リスト7-42〕プログラムstmpsub.asm ①

```
 1: ;*******************************************************************
 2: ;
 3: ;   機   能 :   タイム・スタンプ変更サブルーチン
 4: ;   ファンクション :   1AH(DATアドレスの設定)
 5: ;              3DH(ファイルのオープン)
 6: ;              3EH(ファイルのクローズ)
 7: ;              4EH(一致するファイルの検索)
 8: ;              4FH(つぎのファイルの検索)
 9: ;              57H(タイム・スタンプの参照/変更)
10: ;   生   成 :   masm /ML stmpsub;
11: ;
12: ;*******************************************************************
13:                .MODEL  SMALL, C
14:                .CODE
15: ;*******************************************************************
16: ;
17: ;   構造体 :   _STMP
18: ;   機   能 :   タイム・スタンプ・データ構造の定義
19: ;
20: ;*******************************************************************/
21: _STMP          STRUC
22: time           DW        ?
23: date           DW        ?
24: _STMP          ENDS
25:
26: ;*******************************************************************
27: ;
28: ;   ルーチン名 :   func_1a
29: ;   機   能 :   DTAアドレス設定
30: ;   func     :   1aH(ディスク転送アドレスの設定)
31: ;   入   力   arg1  ・・・DTAへのポインタ
32: ;   出   力 :   なし
33: ;
34: ;*******************************************************************
35: func_1a        PROC      arg1:PTR
36:                mov       dx, arg1
37:                mov       ah, 1Ah                   ;ファンクション1AH
38:                int       21h
39:                ret
40: func_1a        ENDP
41:
42: ;*******************************************************************
43:
44: ;   ルーチン名 :   func_3d
45: ;   機   能 :   ファイルをオープンしハンドルを返す
46: ;   func     :   3DH(ファイルのオープン)
47: ;   入   力   arg1  ・・・ファイル名へのポインタ
48: ;              arg2  ・・・モード (0:read 1:write 2:read/write)
49: ;   出   力 :   AX    ・・・エラーなし:  ファイル ハンドル
50: ;                          エラーあり:  エラー コード(Bit15  1)
51: ;
52: ;*******************************************************************
53: func_3d        PROC      arg1:PTR, arg2:WORD
54:                mov       dx, arg1
55:                mov       ax, arg2
56:                mov       ah, 3Dh                   ;ファンクション3DH
57:                int       21h
58:                jnc       no_error
59:                or        ax, 8000h
60: no_error:
```

〔リスト7-42〕プログラム stmpsub.asm ②

```
 61:            ret
 62: func_3d    ENDP
 63:
 64: ;*****************************************************************
 65: ;
 66: ;      ルーチン名 :    func_3e
 67: ;      機   能 :     ファイルをクローズする
 68: ;      func   :     3EH(ファイルのクローズ)
 69: ;      入   力 :     arg1   ･･･ ファイル・ハンドル
 70: ;      出   力 :     AX         エラーなし :   0
 71: ;                               エラーあり :   エラー・コード
 72: ;
 73: ;*****************************************************************
 74: func_3e    PROC        arg1:WORD
 75:            mov         bx, arg1
 76:            mov         ah, 3Eh                  ;ファンクション3EH
 77:            int         21h
 78:            jc          error
 79:            xor         ax, ax
 80: error:
 81:            ret
 82: func_3e    ENDP
 83:
 84: ;*****************************************************************
 85: ;
 86: ;      ルーチン名 :    func_4e
 87: ;      機   能 :     一致するファイルを検索し結果をDTAに返す
 88: ;      func   :     4EH(一致するファイルの検索)
 89: ;      入   力 :     arg1   ･･･ ファイル名へのポインタ
 90: ;                   arg2   ･･･ 属性
 91: ;      出   力 :     AX         エラーなし :   0
 92: ;                               エラーあり :   エラー・コード
 93: ;
 94: ;*****************************************************************
 95: func_4e    PROC        arg1:PTR, arg2:WORD
 96:            mov         dx, arg1
 97:            mov         cx, arg2
 98:            mov         ah, 4Eh                  ;ファンクション4EH
 99:            int         21h
100:            jc          error
101:            xor         ax, ax
102: error:
103:            ret
104: func_4e    ENDP
105:
106: ;*****************************************************************
107: ;
108: ;      ルーチン名 :    func_4f
109: ;      機   能 :     つぎに一致するファイルを検索し結果をDTAに返す
110: ;      func   :     4FH(つぎのファイル検索)
111: ;      入   力 :     なし
112: ;      出   力 :     AX         エラーなし :   0
113: ;                               エラーあり :   エラー・コード
114: ;
115: ;*****************************************************************
116: func_4f    PROC
117:            mov         ah, 4Fh                  ;ファンクション4FH
118:            int         21h
119:            jc          error
120:            xor         ax, ax
121: error:
122:            ret
123: func_4f    ENDP
124:
125: ;*****************************************************************
126: ;
127: ;      ルーチン名 :    stmp_read
128: ;      機   能 :     タイム・スタンプを読み出し構造体に返す
129: ;      func   :     57H(タイム・スタンプの参照/変更)
130: ;      入   力 :     arg1   ･･･ ファイル・ハンドル
131: ;                   arg2   ･･･ 構造体へのポインタ
132: ;      出   力 :     AX         エラーなし :   0
133: ;                               エラーあり :   エラー・コード
134: ;
135: ;*****************************************************************
```

〔リスト7-42〕プログラム stmpsub.asm ③

```
136: stmp_read    PROC     arg1:WORD, arg2:PTR
137:              mov      bx, arg1
138:              mov      al, 00h                ;参照
139:              mov      ah, 57h                ;ファンクション57H
140:              int      21h
141:              jc       error
142:              mov      bx, arg2
143:              mov      [bx.date], dx
144:              mov      [bx.time], cx
145:              xor      ax, ax
146: error:
147:              ret
148: stmp_read    ENDP
149:
150: ;*************************************************************************
151: ;
152: ;      ルーチン名 :   stmp_write
153: ;      機　　能 :   構造体に設定されたタイム・スタンプを書き出す
154: ;      func     :   57H(タイム・スタンプの参照/変更)
155: ;      入　　力 :   arg1  ･･･ファイル・ハンドル
156: ;                   arg2  ･･･構造体へのポインタ
157: ;      出　　力 :   AX    ･･･エラーなし:   0
158: ;                            エラーあり:   エラー・コード
159: ;
160: ;*************************************************************************
161: stmp_write   PROC     arg1:WORD, arg2:PTR
162:              mov      bx, arg2
163:              mov      cx, [bx.time]
164:              mov      dx, [bx.date]
165:              mov      bx, arg1
166:              mov      al, 01h                ;書き出し
167:              mov      ah, 57h                ;ファンクション57H
168:              int      21h
169:              jc       error
170:              xor      ax, ax
171: error:
172:              ret
173: stmp_write   ENDP
174:              END
```

◆ **実行サンプル**

リスト7-43は，タイム・スタンプの変更プログラムstmp.exeの実行例を示しています．

① dirコマンドを用いてカレント・ディレクトリ内にある拡張子.cのファイルを確認する．

② ここで，タイム・スタンプに注目．

③ プログラムstmp.exeを用いて拡張子.cの日付を1989年1月1日に，時刻を12時00分に変更する．ここで，分と秒の指定が省略されているように，年月日や時分秒の全部または一部が省略可能である．また，ワイルド・カードも自由に使用できる．

④ 再びdirコマンドを用いて拡張子.cのファイルを確認する．

⑤ ここで，タイム・スタンプが指定したとおりに変更されている．

⑥ 次に，dirコマンドを用いてサブ・ディレクトリの確認を行う．

⑦ 同様に，dirコマンドを用いてサブ・ディレクトリsourceの中にある拡張子.asmのファイルを確認する．

⑧ ここでタイム・スタンプに注目．

⑨ プログラムstmp.exeを用いて，サブ・ディレクトリの中にある拡張子.asmのタイム・スタンプのうち，日付を1989年1月2日に，時刻を1時10分に指定して変更する．ここで示したように，プログラムstmp.exeでは，ファイル名としてディレクトリ名を含んだパス名も使用できる．

⑩ 再びdirコマンドで指定したファイルのタイム・スタンプが変更されたかを確認する．

⑪ 指定されたとおりに変更されている．

⑫ 次に，dirコマンドを用いて拡張子.comのファイルを確認する．ここで，拡張子.comのファイルは存在しない．

⑬ プログラムstmp.exeに対して存在しないファイル名.comを指定して起動する．

⑭ すると，エラー・メッセージが表示される．

〔リスト7-43〕プログラム stmp.exe の実行例 ①

```
R>dir *.c ⏎… 拡張子.cのファイルを確認①

 ドライブ R: のディスクのボリュームラベルは [IOS-10     ]
 ディレクトリは R:¥WK¥TEST

CTRL     C       1364  88-12-07   9:04  ←(タイム・スタンプに注目②)
TEE      C        402  88-11-28  15:02
FATAL    C       3620  88-12-07  11:42



HEAD     C        269  88-12-01  11:27
DDUMP    C       5412  88-12-07  11:42
       11 個のファイルがあります.
    40960 バイトが使用可能です.

R>stmp *.c -d89-1-1 -t12: ⏎….cファイルのタイム・スタンプを変更③

 *** タイム・スタンプ変更プログラム Ver.1.1 ***

R>dir *.c  ⏎….cファイルの確認④

 ドライブ R: のディスクのボリュームラベルは [IOS-10     ]
 ディレクトリは R:¥WK¥TEST

CTRL     C       1364  89-01-01  12:00  ←(タイム・スタンプが変更された⑤)
TEE      C        402  89-01-01  12:00
FATAL    C       3620  89-01-01  12:00



HEAD     C        269  89-01-01  12:00
DDUMP    C       5412  89-01-01  12:00
       11 個のファイルがあります.
    40960 バイトが使用可能です.

R>dir *.  ⏎… サブディレクトリの確認⑥

 ドライブ R: のディスクのボリュームラベルは [IOS-10     ]
 ディレクトリは R:¥WK¥TEST

.            <DIR>     88-12-09   9:36
..           <DIR>     88-12-09   9:36
SOURCE       <DIR>     88-12-09  10:31
SOURCE1      <DIR>     88-12-09  10:32
        4 個のファイルがあります.
    40960 バイトが使用可能です.

R>dir source¥*.asm  ⏎… サブディレクトリ source 内の .asm ファイルの確認⑦

 ドライブ R: のディスクのボリュームラベルは [IOS-10     ]
 ディレクトリは R:¥WK¥TEST¥SOURCE

DIRSORT  ASM     4521  85-12-04  22:14  ←(タイム・スタンプに注目⑧)
CTRLSUB  ASM     2237  88-12-07   8:29
FATALSUB ASM     4781  88-12-07   8:36



FCB      ASM     3452  85-11-26  10:49
HANDLE   ASM     5779  85-11-26  10:57
CHILD    ASM     5896  88-12-06  10:55
       15 個のファイルがあります.
    38912 バイトが使用可能です.

R>stmp source¥*.asm -d89-1-2 -t1:10:  ⏎… ディレクトリ source 内の .asm のタイム・スタンプを変更⑨

 *** タイム・スタンプ変更プログラム Ver.1.1 ***

R>dir source¥*.asm  ⏎… サブディレクトリ source 内の .asm ファイルの確認⑩
 ドライブ R: のディスクのボリュームラベルは [IOS-10     ]
 ディレクトリは R:¥WK¥TEST¥SOURCE
```

〔リスト7-43〕プログラム stmp.exe の実行例 ②

```
DIRSORT  ASM     4521  89-01-02   1:10  ←(タイム・スタンプが変更された ⑪)
CTRLSUB  ASM     2237  89-01-02   1:10
FATALSUB ASM     4781  89-01-02   1:10



FCB      ASM     3452  89-01-02   1:10
HANDLE   ASM     5779  89-01-02   1:10
CHILD    ASM     5896  89-01-02   1:10
      15 個のファイルがあります.
   38912 バイトが使用可能です.

R>dir *.com  ⏎….com ファイルの確認 ⑫

 ドライブ R: のディスクのボリュームラベルは [IOS-10   ]
 ディレクトリは R:¥WK¥TEST

ファイルが見つかりません.

R>stmp *.com -d89-1-3 -t2:20:00  ⏎… 存在しないファイル名の指定 ⑬

 *** タイム・スタンプ変更プログラム Ver.1.1 ***

ファイルがありません. ←(エラー・メッセージ ⑭)

R>
```

デバイス・データを調べる

リスト7-44(gdev.c)および**リスト7-45**(gdevsub.asm)は，パラメータで指定されたファイル・ハンドルをもつデバイス，あるいは指定されたデバイス名をもつデバイス，または指定されたファイルのあるドライブのデバイス・データを調べ，その表示を行うプログラム gdev.exe のソース・リストです．

● gdev.c

リスト7-44はプログラム gdev.exe の C ソース部分です．同リストにおいて，関数 main では最初にプログラム・タイトルを表示したのち，コマンド・ライン・パラメータの解析を行います．

パラメータの解析では，パラメータの個数を調べ，もしパラメータにファイル・ハンドルやパス名，あるいはデバイス名などが指定されていなければ，プログラム gdev.exe の使用方法を表示してエラー・ストップします．コマンド・ライン・パラメータの解析が終わったら関数 chk_dev によってデバイス・データを調べてその情報を表示します．

関数 chk_dev は，ポインタ ptr で指定されたデバイス名やパス名(ファイル名)，あるいはファイル・ハンドル(標準でオープンされているデバイスに限定)をもつデバイスのデバイス・データを調べます．ポインタ ptr で指定された文字列が数字の場合はファイル・ハンドルなので，そのファイル・ハンドルが標準入出力デバイスに対応しているかどうかを調べ，標準入出力のファイル・ハンドルでなければエラー表示してプログラムを終了します．

もし，ポインタ ptr で指定された文字列が，CON や AUX など標準入出力のデバイス名である場合は，それぞれのデバイス名に対応したファイル・ハンドルに変換し，そのファイル・ハンドルを変数 n に格納します．

〔リスト7-44〕プログラム gdev.c ①

```
 1: /**************************************************************
 2: *
 3: *   機　　能：   デバイス・データの表示
 4: *   サ　ブ：    gdevsub.asm
 5: *   生　　成：   masm /ML gdevsub;
 6: *             cl -J -AS gdev.c gdevsub
 7: *   使用方法：   gdev [<ファイル名> ¦ <デバイス名>]
 8: *
 9: ***************************************************************/
10: #include    <stdio.h>
11: #include    <ctype.h>
```

〔リスト7-44〕プログラム gdev.c ②

```
12: #include     <string.h>
13: #include     <stdlib.h>
14:
15: void main (int, char **);
16: void chk_dev (char *);
17: void dev_msg (int);
18: int  func_3d (char *);
19: int  func_4400 (int);
20: /**********************************************************
21: *
22: *    関数名 :   main (argc, argv)
23: *    機  能 :   コマンドライン・パラメータの解析
24: *    入  力 :   int argc          ・・・・・ コマンドライン・パラメータの数
25: *               char *argv[]      ・・・・・ パラメータ文字列へのポインタ
26: *    出  力 :   なし
27: *
28: **********************************************************/
29: void main (argc, argv)
30: int argc;
31: char **argv;
32: {
33:     char *ptr;
34:
35:     printf ("¥n *** デバイス・チェック・プログラム Ver.1.1 ***¥n¥n");
36:     if (argc != 2) {
37:         printf ("使用法 : gdev < パス名 ¦ ファイル・ハンドル >¥n");
38:         exit (1);
39:     }
40:     ptr = *++argv;
41:     chk_dev (ptr);
42:     printf ("¥n");
43:     exit (0);
44: }
45:
46: /**********************************************************
47: *
48: *    関数名 :   chk_dev (ptr)
49: *    機  能 :   デバイス・データの取得
50: *    入  力 :   ptr ・・・ パス名へのポインタ
51: *    出  力 :   なし
52: *
53: **********************************************************/
54: void chk_dev (ptr)
55: char *ptr;
56: {
57:     int n, st;
58:
59:     if (isdigit (*ptr)) {
60:         if ((n = atoi (ptr)) > 4) {
61:             printf ("ファイル・ハンドルは標準入出力を指定して下さい.¥n");
62:             printf ("標準入力 (CON)           ・・・・  0¥n");
63:             printf ("標準出力 (CON)           ・・・・  1¥n");
64:             printf ("標準エラー出力 (CON)     ・・・・  2¥n");
65:             printf ("外部入出力 (AUX)         ・・・・  3¥n");
66:             printf ("標準プリンタ (PRN)       ・・・・  4¥n");
67:             exit (2);
68:         }
69:     } else {
70:         if (! strcmpi (ptr, "CON")) {
71:             n = 0;
72:         } else if (! strcmpi (ptr, "AUX")) {
73:             n = 3;
74:         } else if (! strcmpi (ptr, "PRN")) {
75:             n = 4;
76:         } else if ((n = func_3d (ptr)) & 0x8000) {
77:             switch (n) {
78:             case 0x8001:
79:                 printf ("無効なファンクション・コードです.¥n"); break;
80:             case 0x8002:
81:                 printf ("該当ファイルがありません.¥n"); break;
82:             case 0x8003:
83:                 printf ("パス名が無効です.¥n"); break;
84:             case 0x8004:
85:                 printf ("オープンされているファイルが多すぎます.¥n"); break;
86:             case 0x8005:
87:                 printf ("アクセスが拒否されました.¥n"); break;
```

〔リスト7-44〕プログラム gdev.c ③

```
 88:             default:
 89:                 printf ("不正なアクセスです.¥n"); break;
 90:             }
 91:             printf ("¥n");
 92:             exit (3);
 93:         }
 94:     }
 95:     st = func_4400 (n);
 96:     if (st & 0x8000) {
 97:         printf ("IOCTL 読み出しエラーです.¥n");
 98:         exit (4);
 99:     }
100:     dev_msg (st);
101: }
102:
103: /*************************************************************************
104:  *
105:  *    関数名 :  dev_msg (st)
106:  *    機  能 :  デバイス・データの表示
107:  *    入  力 :  st  :  デバイス・データ
108:  *    出  力 :  なし
109:  *
110:  *************************************************************************/
111: void dev_msg (st)
112: int st;
113: {
114:     printf ("指定されたデバイスは");
115:     if (st & 0x0080) {
116:         printf ("キャラクタ・デバイスです.¥n");
117:     } else {
118:         printf ("ブロック・デバイスです ¥n");
119:     }
120:     printf ("デバイス・データ (16進)              ......  '%04X'¥n", st);
121:     printf ("I/Oコントロール・データの送受         ......  ");
122:     if (st & 0x0400) {
123:         printf ("可能¥n");
124:     } else {
125:         printf ("不可能¥n");
126:     }
127:     printf ("EOFの入力                          ......  ");
128:     if (st & 0x0040) {
129:         printf ("不可能¥n");
130:     } else {
131:         printf ("可能¥n");
132:     }
133:     printf ("コントロール・キャラクタのチェック     ......  ");
134:     if (st & 0x0020) {
135:         printf ("しない¥n");
136:     } else {
137:         printf ("する¥n");
138:     }
139:     printf ("INT 29Hで文字の出力                 ......  ");
140:     if (st & 0x0010) {
141:         printf ("可能¥n");
142:     } else {
143:         printf ("不可能¥n");
144:     }
145:     if (st & 0x0080) {
146:         printf ("デバイス                           ......  ");
147:         switch (st & 0x000F) {
148:         case 1:
149:             printf ("標準入力¥n"); break;
150:         case 2:
151:             printf ("標準出力¥n"); break;
152:         case 3:
153:             printf ("標準入出力¥n"); break;
154:         case 4:
155:             printf ("NULデバイス¥n"); break;
156:         case 8:
157:             printf ("クロック・デバイス¥n"); break;
158:         }
159:     } else {
160:         printf ("ドライブ番号 (ドライブ名)                   ......  %c¥n",
161:                 (st & 0x001F) + 'A');
162:     }
163: }
```

もし，ポインタ ptr で指定された文字列が，パス名(ファイル名やディレクトリ名)の場合には，関数(サブルーチン)func_3d を用いてファイル・オープンを行い，その結果得られたファイル・ハンドルを変数 n に格納します．

ここで，関数 func_3d から返された値のビット 15 がセットされている場合は，その値はエラー・コードを表しているので，それぞれのエラー・コードに対応したエラー・メッセージを表示してエラー・ストップします．

ファイル・ハンドルが正常に返されたら，次に関数 func_4400 を用いてそのファイル・ハンドルをもつデバイスのデバイス・データの読み出しを行います．ここでも，関数 func_4400 でエラーが発生している場合は，ステータスとして返す値のビット 15 がセットされて返されるのでエラー表示してストップします．

デバイス・データが正常に読み出されたら，関数 dev_msg を用いてそのデバイス・データの情報を表示します．

関数 dev_msg では，引数として渡されたデバイス・データの各情報に基づいて，それぞれに対応したメッセージの表示を行います(図6-7：176 ページ)．

● gdevsub.asm

リスト7-45 はプログラム gdev.exe のアセンブリ・ソース部分です．同リストにおいて，サブルーチン(関数)func_3d は，引数 arg1 で渡されたパス名とファンクション 3DH を用いて，ファイル・ハンドルのオープンを行い，その結果得られたファイル・ハンドルを AX レジスタに返します．ここで，もしファンクション 3DH においてエラーが発生した場合は，そのエラー・ステータスが AX レジスタに返されるので，ビット 15

〔リスト7-45〕
プログラム
gdevsub.asm

```
 1: ;*************************************************************************
 2: ;
 3: ;   機    能 :   デバイス・チェック・サブルーチン
 4: ;   ファンクション :   3DH(ファイル・ハンドルのオープン)
    ;                  4400H(デバイス・データの取得)
 5: ;
 6: ;   生    成 :   masm /ML gdevsub;
 7: ;
 8: ;*************************************************************************
 9:                 .MODEL  SMALL, C
10:                 .CODE
11: ;*************************************************************************
12: ;
13: ;   ルーチン名 :   func_3d
14: ;   機    能 :   ファイル・ハンドルの取得
15: ;   func     :   3DH(ファイルのオープン)
16: ;   入    力 :   arg1   ・・・ パス名へのポインタなし
17: ;   出    力 :   AX     ・・・ ファイル・ハンドル
18: ;                       エラーの場合 : Bit15をセット
19: ;
20: ;*************************************************************************
21: func_3d      PROC     arg1:WORD
22:              mov      dx, arg1
23:              xor      al, al
24:              mov      ah, 3Dh                 ;ファンクション 3DH
25:              int      21h
26:              jnc      no_err
27:              or       ax, 8000h
28: no_err:
29:              ret
30: func_3d      ENDP
31:
32: ;*************************************************************************
33: ;
34: ;   ルーチン名 :   func_4400
35: ;   機    能 :   デバイス・データの読み出し
36: ;   func     :   4400H(デバイス・データ取得)
37: ;   入    力 :   arg1   ・・・ ファイル・ハンドル
38: ;   出    力 :   AX     ・・・ デバイス・データ
39: ;                       エラーの場合 : Bit15をセット
40: ;
41: ;*************************************************************************
42: func_4400    PROC     arg1:WORD
43:              mov      bx, arg1
44:              mov      ax, 4400h                         ;ファンクション 4400H
45:              int      21h
46:              jc       error
47:              and      dx, 7FFFh
48: error:
49:              mov      ax, dx
50:              ret
51: func_4400    ENDP
52:              END
```

をセットして返すことによって，Cソース内でファイル・ハンドルとエラー情報の違いを判定できるようにしています．

サブルーチン func_4400 は，引数 arg1 で指定されたファイル・ハンドルをもつデバイスや，そのファイル・ハンドルで指定されたファイルの入っているドライブのデバイス・データをファンクション 4400H を用いて調べて，その得られたデバイス・データを AX レジスタに返します．ここでも，もしファンクション 4400H においてエラーが発生した場合は，AX レジスタのビット 15 をセットして返します．

◆ **生成方法**

プログラム gdev.exe は，次の手順で分割アセンブル/コンパイルして作成します．

```
masm /ML gdevsub;
cl -J -As gdev.c gdevsub
```

◆ **実行サンプル**

リスト7-46 は，デバイス・データを調べて表示するプログラム gdev.exe の実行例を示しています．

① dir コマンドを用いてドライブ Y のルート・ディレクトリを確認する．ここではサブ・ディレクトリ ¥pro のみが存在している．

② さらに，dir コマンドを用いてそのサブ・ディレクトリ ¥pro を確認する．

③ ここで，これからアクセスしようとするファイル名の存在に注目．

④ まず，プログラム gdev.exe に対しパラメータとしてドライブ名を与えて起動する．

⑤ するとドライブ名だけではアクセスできない旨のエラー・メッセージが表示される．

⑥ 次に，プログラム gdev.exe に対し，パラメータとしてディレクトリ名を与えて起動する．

⑦ ここでも，アクセスが拒否されエラー・メッセージが表示される．

⑧ 次に，プログラム gdev.exe に対してパス名(ファイル名)を与えて起動する．

⑨ すると，その指定されたファイルの入っているドライブ(すなわちドライブ Y)のディレクトリが表示される(ドライブ Y は，1M バイト・フロッピ・ディスク)

〔リスト7-46〕プログラム gdev.exe の実行例 ①

```
R>dir y: ⏎… ドライブYのカレント・ディレクトリ(ルート・ディレクトリ)の確認①

 ドライブ Y: のディスクのボリュームラベルはありません．
 ディレクトリは Y:¥

PRO          <DIR>     88-12-06   16:17
        1 個のファイルがあります．
  1080320 バイトが使用可能です．

R>dir y:¥pro ⏎… ドライブYのディレクトリ pro を確認②

 ドライブ Y: のディスクのボリュームラベルはありません．
 ディレクトリは Y:¥PRO

.            <DIR>     88-12-06   16:17
..           <DIR>     88-12-06   16:17
CTRL     C        1364 88-12-05   13:27 ← アクセスしようとするファイル③
TEE      C         402 88-11-28   15:02
FATAL    C        3614 88-12-05   13:27
                  .
                  .
                  .
DIV      SMP       227 88-12-05   17:52
CHILD    SMP      2021 88-12-06   10:53
MALOC    SMP       969 88-12-06   16:14
       66 個のファイルがあります．
  1080320 バイトが使用可能です．

R>gdev y: ⏎… ドライブ名の指定④

  *** デバイス・チェック・プログラム Ver.1.1 ***
該当ファイルがありません． ← エラー・メッセージ⑤

R>gdev y:¥pro ⏎… ディレクトリ名の指定⑥

  *** デバイス・チェック・プログラム Ver.1.1 ***
 アクセスが拒否されました． ← エラー・メッセージ⑦
```

〔リスト7-46〕プログラム gdev.exe の実行例 ②

```
R>gdev y:¥pro¥ctrl.c  ⏎…フル・パス名(ファイル名)の指定⑧

 *** デバイス・チェック・プログラム Ver.1.1 ***

指定されたデバイスはブロック・デバイスです.
デバイス・データ(16進)         ……   '0042'  ← 指定されたファイルの入っているデバイス
                                              (ドライブ)のデバイス・データ⑨
                                              (1M バイト・フロッピ)
I/Oコントロール・データの送受    ……   不可能
EOFの入力                       ……   不可能
コントロール・キャラクタのチェック ……   する
INT 29Hで文字の出力              ……   不可能
ドライブ番号(ドライブ名)          ……   C
                       ドライブYはドライブCをASSIGNした論理ドライブ⑩

R>dir h:  ⏎… ドライブHのカレント・ディレクトリを確認⑪

 ドライブ H: のディスクのボリュームラベルは HD1
 ディレクトリは H:¥WK1¥WK

.            <DIR>      88-11-08  15:16
..           <DIR>      88-11-08  15:16
MADD     BAT        18  88-11-28  10:58
CTRL     C        1364  88-12-07   9:04 ← アクセスしようとするファイル
TEE      C         402  88-11-28  15:02
     .
     .
     .
MEDIA    EXE     10627  88-12-07  17:08
GDEVSUB  ASM      1544  88-12-07  17:11
GDEV     C        4287  88-12-07  17:15
      109 個のファイルがあります.
  6078464 バイトが使用可能です.

R>gdev h:ctrl.c  ⏎… ファイル名の指定⑫

 *** デバイス・チェック・プログラム Ver.1.1 ***

指定されたデバイスはブロック・デバイスです.
デバイス・データ(16進)         ……   '0040'  ← ファイルの入っているドライブの
                                              デバイス・データ⑬
                                              (20M バイト・ハード・ディスク)
I/Oコントロール・データの送受    ……   不可能
EOFの入力                       ……   不可能
コントロール・キャラクタのチェック ……   する
INT 29Hで文字の出力              ……   不可能
ドライブ番号(ドライブ名)          ……   A   ← ドライブHはドライブAをASSIGN⑭

R>dir  ⏎… カレント・ドライブのカレント・ディレクトリを確認⑮
                                        カレント・ドライブはドライブR⑯
 ドライブ R: のディスクのボリュームラベルは [IOS-10    ]
 ディレクトリは R:¥WK

.            <DIR>      88-12-08   8:12
..           <DIR>      88-12-08   8:12
MADD     BAT        18  88-11-28  10:58
TEE      C         402  88-11-28  15:02
CTRL     C        1364  88-12-07   9:04 ← アクセスしようとするファイル
     .
     .
     .
GDEV     MAP      8516  88-12-08  10:16
GDEV     EXE     13288  88-12-08  10:16
CLOSE    BAT       102  88-12-08  10:16
      124 個のファイルがあります.
   636928 バイトが使用可能です.
```

⑩ ドライブYは，ドライブCがASSIGNされた論理ドライブであることも確認できる.

⑪ 次に，ドライブH(ハード・ディスク)のファイルをアクセスするためにサブ・ディレクトリ ¥wk1¥wk を確認する.

⑫ そのディレクトリにあるファイル名を指定してプログラム gdev.exe を起動する.

⑬ すると，指定されたファイルの入っているドライブ(すなわちハード・ディスク)のデバイス・データが表示される.

⑭ ここでも，ドライブHはドライブAがASSIGNされた論理ドライブ・ディレクトリあることが確認できる.

⑮ 次に，dir コマンドを用いてカレント・ドライブの

〔リスト7-46〕プログラム gdex.exe の実行例 ③

```
R>gdev ctrl.c  ⏎… ファイル名の指定⑰

 *** デバイス・チェック・プログラム Ver.1.1 ***

指定されたデバイスはブロック・デバイスです.
デバイス・データ（16進）            .......  '0043'
I／Oコントロール・データの送受     .......  不可能
EOFの入力                          .......  不可能
コントロール・キャラクタのチェック  .......  する
INT 29Hで文字の出力                .......  不可能
ドライブ番号（ドライブ名）          .......  D

R>gdev 0  ⏎… ファイル・ハンドル（標準入力）の指定⑳

 *** デバイス・チェック・プログラム Ver.1.1 ***

指定されたデバイスはキャラクタ・デバイスです.
デバイス・データ（16進）            .......  '00D3'
I／Oコントロール・データの送受     .......  不可能
EOFの入力                          .......  不可能
コントロール・キャラクタのチェック  .......  する
INT 29Hで文字の出力                .......  可能
デバイス                                     標準入出力

R>gdev 2  ⏎…ファイル・ハンドル（標準エラー出力）の指定㉒

 *** デバイス・チェック・プログラム Ver.1.1 ***

指定されたデバイスはキャラクタ・デバイスです.
デバイス・データ（16進）            .......  '00D3'
I／Oコントロール・データの送受     .......  不可能
EOFの入力                          .......  不可能
コントロール・キャラクタのチェック  .......  する
INT 29Hで文字の出力                .......  可能
デバイス                                     標準入出力

R>gdev aux  ⏎… デバイス名の指定㉓

 *** デバイス・チェック・プログラム Ver.1.1 ***

指定されたデバイスはキャラクタ・デバイスです.
デバイス・データ（16進）            .......  '00C0'
I／Oコントロール・データの送受     .......  不可能
EOFの入力                          .......  不可能
コントロール・キャラクタのチェック  .......  する
INT 29Hで文字の出力                .......  不可能
デバイス

R>gdev prn  ⏎… デバイス名の指定㉕

 *** デバイス・チェック・プログラム Ver.1.1 ***

指定されたデバイスはキャラクタ・デバイスです.
デバイス・データ（16進）            .......  '00C0'
I／Oコントロール・データの送受     .......  不可能
EOFの入力                          .......  不可能
コントロール・キャラクタのチェック  .......  する
INT 29Hで文字の出力                .......  不可能
デバイス

R>
```


カレント・ディレクトリを確認する.

⑯ ここで，カレント・ドライブはドライブR(RAMディスク)であることが確認できる.

⑰ 次に，プログラム gdev.exe に対しカレント・ドライブ上のファイル名を与えて起動する.

⑱ すると，その指定されたファイルの入っているドライブ(すなわちRAMディスク)のデバイス・データが表示される.

⑲ ここでも，ドライブRはドライブDがASSIGNされた論理ドライブであることが確認できる.

⑳ 次に，プログラム gdev.exe に対しパラメータとしてファイル・ハンドルの0(標準入力)を与えて起動する.

㉑ すると，そのファイル・ハンドルに対応したデバ

イス(CON)のデバイス・データが表示される.

㉒ 同様に，プログラム gdev.exe に対しパラメータとしてファイル・ハンドルの 2 (標準エラー出力)を与えて起動する．ここでも，標準エラー出力のデバイスは CON なので，そのデバイス・データが表示される.

㉓ 次に，プログラム gdev.exe に対しパラメータとして直接デバイス名 AUX(補助入出力)を与えて起動する.

㉔ すると，指定したとおりデバイス AUX のデバイス・データが表示される.

㉕ 同様に，デバイス PRN(プリンタ出力)のデバイス・データを確認する.

㉖ デバイス PRN もデバイス AUX と同様のデバイス・データであることが確認できる.

メディア交換の可能性を調べる

リスト7-47(media.c)と**リスト7-48**(mediasub.asm)は，指定されたドライブのメディア交換が可能かどうかを調べて表示するプログラム media.exe のソース・リストです.

● media.c

リスト7-47はプログラム media.exe の C ソース部分です．同リストにおいて，関数 main では最初にプログラム・タイトルを表示したのち，コマンド・ライン・パラメータの解析を行います.

パラメータの解析では，ドライブの指定があった場合には，そのドライブ名をドライブ番号に変換します．もし，ドライブ名の指定がない場合には，関数(サブルーチン) func_19 を用いてカレント・ドライブ番号の読み出しを行って変数 drv_num に格納しておきます．ドライブ番号の設定が終わったら，関数 chk_media を用いてメディア交換可能性の有無を表示します.

関数 chk_media では，引数 n で指定されたドライブ番号からドライブ名に変換して表示したのち，関数 func_4408 を用いて，ドライブ番号 n で指定したドライブのメディア交換が可能かどうかを調べ，その関数 func_4408 から返されたステータスにしたがってメッセージの表示を行います.

● mediasub.asm

リスト7-48はプログラム media.exe のアセンブリ・ソース部分です．同リストにおいて，サブルーチン(関数) func_19 は，ファンクション 19H を用いてカレント・ドライブ番号の取得を行い，そのドライブ番号を AX レジスタに返します.

サブルーチン func_4408 は，ファンクション 4408H を用いて，引数 arg1 で指定されたドライブのメディア交換の可能性を調べ，そのステータスを AX レジスタに返します．ここで，もしファンクション 4408H でエラーが発生した場合は，AX レジスタのビット 15 をセットして返します.

● 8086 vs 68000(その5) IBM が 68000 を採用 ●

「ついに IBM-PC に 68000 が採用されました．IBM は，これまでのインテル社との恋愛関係に見切りをつけ，今度はモトローラ社と手を結ぶことになりました.

これに同調して，日本においても NEC をはじめとするコンピュータ・メーカ各社が，パソコン CPU に 68000 を採用することを決定しました.

全世界のソフトウェア技術者たちは，『魔のセグメント』から解放され，インテル得意技の『特殊な汎用』レジスタも意識することがなくなったため，一様に明るい表情となって大歓迎しています.

これによって，アプリケーション・ソフトの質や量の向上，および開発期間の短縮など，非常に大きな波及効果が期待できることになりそうです．…」

アッ，プログラミングに疲れてついウトウトしてしまった．夢か….

〔リスト7-47〕プログラム media.c

```
 1: /***********************************************************************
 2: *
 3: *   機    能 :  メディア交換の可能性チェック
 4: *   サ    ブ :  mediasub.asm
 5: *   生    成 :  masm /ML mediasub;
 6: *               cl -J -AS media.c mediasub
 7: *   使用方法 :  media [<d:>]
 8: *
 9: ***********************************************************************/
10: #include    <stdio.h>
11:
12: void main (int, char **);
13: void chk_media (int);
14: int  func_19 (void);         /* カレント・ドライブ番号の読み出し */
15: int  func_4408 (int);        /* メディア交換可能性のチェック   */
16: /**********************************************************************
17: *
18: *   関数名 :  main (argc, argv)
19: *   機  能 :
20: *   入  力 :  int argc        ・・・・  コマンドライン・パラメータの数
21: *             char *argv[]    ・・・・  パラメータ文字列へのポインタ
22: *   出  力 :  なし
23: *
24: **********************************************************************/
25: void main (argc, argv)
26: int argc;
27: char **argv;
28: {
29:     int drv_num;
30:
31:     printf ("¥n *** メディア・チェック・プログラム Ver.1.1 ***¥n¥n");
32:     drv_num = 0;
33:     if (argc != 1) {
34:         drv_num = toupper (*++argv) - 'A';
35:     } else {
36:         drv_num = func_19 ();
37:     }
38:     chk_media (drv_num);
39:     printf ("¥n");
40:     exit (0);
41: }
42:
43: /**********************************************************************
44: *
45: *   関数名 :  chk_media (n)
46: *   機  能 :  メディア交換可能性チェック
47: *   入  力 :  n ・・・ ドライブ番号 ( 00H = カレント , 01H = A, ・・・)
48: *   出  力 :  なし
49: *
50: **********************************************************************/
51: void chk_media (n)
52: int n;
53: {
54:     int st;
55:
56:     printf ("ドライブ %c の", n + 'A');
57:     st = func_4408 (n + 1);
58:     switch (st) {
59:     case 0x00:
60:         printf ("メディアは交換が可能です.¥n");
61:         break;
62:
63:     case 0x01:
64:         printf ("メディアの交換は不可能です.¥n");
65:         break;
66:
67:     case 0x800F:
68:         printf ("ドライブ指定が不正です.¥n");
69:         break;
70:
71:     default:
72:         printf ("ドライバがメディア・チェックをサポートしていません.¥n");
73:         break;
74:     }
75: }
```

〔リスト7-48〕プログラム mediasub.asm

```
 1: ;******************************************************************
 2: ;
 3: ;    機    能 :   メディア・チェック・サブルーチン
 4: ;    ファンクション :   19H(カレント・ドライブ番号の呼び出し)
 5: ;                  4408H(メディア交換可能性のチェック)
 6: ;    生    成 :   masm /ML mediasub;
 7: ;
 8: ;******************************************************************
 9:              .MODEL   SMALL, C
10:              .CODE
11: ;******************************************************************
12: ;
13: ;    ルーチン名 :   func_19
14: ;    機    能 :   カレント・ドライブ番号の取得
15: ;    func    :   19H(カレント・ドライブ番号の読み出し)
16: ;    入    力 :   なし
17: ;    出    力 :   ax  ・・・ ドライブ番号(00H=A, 01H=B, ・・・)
18: ;
19: ;******************************************************************
20: func_19      PROC
21:              mov      ah, 19h                 ;ファンクション19H
22:              int      21h
23:              xor      ah, ah
24:              ret
25: func_19      ENDP
26:
27: ;******************************************************************
28: ;
29: ;    ルーチン名 :   func_4408
30: ;    機    能 :   メディアのチェック
31: ;    func    :   4408H(メディア交換可能性のチェック)
32: ;    入    力 :   arg1  ・・・ ドライブ番号
33: ;    出    力 :   AX    ・・・ 00H: 交換可能
34: ;                             01H: デバイスがサポートしていない
35: ;                             01H: ドライブ番号が不正
36: ;                             etc: 交換不可能
37: ;
38: ;******************************************************************
39: func_4408    PROC     arg1:WORD
40:              mov      bx, arg1
41:              mov      ax, 4408h               ;ファンクション4408H
42:              int      21h
43:              jnc      no_err
44:              xor      ah, ah
45:              or       ax, 8000h
46: no_err:
47:              ret
48: func_4408    ENDP
49:              END
```

◆ **生成方法**

プログラム media.exe は，次の手順で分割アセンブル/コンパイルして作成します．

　masm /ML mediasub;

　cl －J －As media.c mediasub

◆ **実行サンプル**

リスト7-49 は，メディア交換可能性チェック・プログラム media.exe の実行例を示しています．

① まず，プログラム getd.exe(前出)によってカレント・ドライブ(ドライブ R)のディスク情報を表示して，どのようなドライブであるかを確認する．

② ここで，ドライブ R は ASSIGN された 1.5 M バイトの RAM ディスクであることがわかる．

③ 次に，プログラム media.exe に対してパラメータを与えないで起動する．これによって，カレント・ドライブ(RAM ディスク)のメディア交換の可能性を調べることができる．

④ 当然のことながら，RAM ディスクではメディア交換を行うことができないので，交換できない旨のメッセージが表示される．

⑤ 次に，プログラム getd.exe によってドライブ Y のディスク情報を調べる．

⑥ ここで，ドライブ Y は 1 M バイト・フロッピ・ディスクであることがわかる．

⑦ プログラム media.exe に対して，パラメータとしてドライブ Y を指定して起動する．

〔リスト7-49〕プログラム media.exe の実行例

```
R>getd ⏎… カレント・ドライブのディスク情報①

 *** ディスク情報表示プログラム Ver.1.1 ***

 ドライブ R のディスク情報

FAT ID                          ---  F8              ← 1.5MバイトRAMディスク②
1ドライブあたりのクラスタ数       ---  760 クラスタ
1クラスタあたりのセクタ数         ---  4 セクタ
1セクタあたりのバイト数           ---  512 バイト
ディスク容量                     ---  1556480 バイト
残り容量                         ---  491520 バイト

R>media ⏎… パラメータなしで起動③

 *** メディア・チェック・プログラム Ver.1.1 ***

 ドライブ R のメディアの交換は不可能です.← メッセージ④

R>getd y: ⏎… ドライブYのディスク情報⑤

 *** ディスク情報表示プログラム Ver.1.1 ***

 ドライブ Y のディスク情報

FAT ID                          ---  FE              ← 1Mバイト・フロッピ⑥
1ドライブあたりのクラスタ数       ---  1221 クラスタ
1クラスタあたりのセクタ数         ---  1 セクタ
1セクタあたりのバイト数           ---  1024 バイト
ディスク容量                     ---  1250304 バイト
残り容量                         ---  508928 バイト

R>media y: ⏎… ドライブYの指定⑦

 *** メディア・チェック・プログラム Ver.1.1 ***

 ドライブ Y のメディア交換が可能です.← メッセージ⑧

R>getd h: ⏎… ドライブHのディスク情報⑨

 *** ディスク情報表示プログラム Ver.1.1 ***

 ドライブ H のディスク情報

FAT ID                          ---  FE              ← 20Mバイトのハード・ディスク⑩
1ドライブあたりのクラスタ数       ---  2468 クラスタ
1クラスタあたりのセクタ数         ---  8 セクタ
1セクタあたりのバイト数           ---  1024 バイト
ディスク容量                     ---  20217856 バイト
残り容量                         ---  6144000 バイト

R>media h: ⏎… ドライブH指定⑪

 *** メディア・チェック・プログラム Ver.1.1 ***

 ドライブ R のメディアの交換は不可能です.← メッセージ⑫

R>
```

⑧ すると，ドライブYはフロッピ・ディスクなのでメディア交換が可能な旨のメッセージが表示される．

⑨ 同様に，プログラム getd.exe を用いてドライブHのディスク情報を確認する．

⑩ ドライブHは，20Mバイトのハード・ディスクであることがわかる．

⑪ プログラム media.exe によってドライブHのメディア交換の可能性を調べる．

⑫ 当然のことながらハード・ディスクもメディア交換できない．

7-5 ディレクトリの操作

ここでは，MS-DOSの大きな特徴である階層ディレクトリの操作を行うプログラムを作成します．

サブ・ディレクトリを作成する

リスト7-50(mkd.c)およびリスト7-51(mkdsub.asm)は，サブ・ディレクトリの作成を行うプログラムmkd.exeのソース・リストです．

〔リスト7-50〕プログラム mkd.c

```
 1: /*****************************************************************
 2: *
 3: *   機    能 :  ディレクトリの作成
 4: *   サ    ブ :  mkdsub.asm
 5: *   生    成 :  masm /ML mkdsub;
 6: *               cl -J -AS mkd.c mkdsub
 7: *   使用方法 :  mkd <パス名>
 8: *
 9: ******************************************************************/
10: #include      <stdio.h>
11:
12: void main (int, char **);
13: void mk_dir (char *);
14: int  func_39 (char *);            /* サブ・ディレクトリの作成 */
15: /*****************************************************************
16: *
17: *    関数名 :  main (argc, argv)
18: *    機  能 :  コマンドライン・パラメータの解析
19: *    入  力 :  int argc        コマンドライン・パラメータの数
20: *              char *argv[]    パラメータ文字列へのポインタ
21: *    出  力 :  なし
22: *
23: ******************************************************************/
24: void main (argc, argv)
25: int argc;
26: char **argv;
27: {
28:
29:     printf ("¥n *** ディレクトリ作成プログラム Ver.1.1 ***¥n¥n");
30:     if (argc != 2) {
31:         printf ("使用法: mkd  パス名¥n");
32:         exit (1);
33:     }
34:     mk_dir (*++argv);
35:     printf ("¥n");
36:     exit (0);
37: }
38:
39: /*****************************************************************
40: *
41: *    関数名 :  mk_dir (ptr)
42: *    機  能 :  ディレクトリの作成
43: *    入  力 :  ptr ･･･ パス名へのポインタ
44: *    出  力 :  なし
45: *
46: ******************************************************************/
47: void mk_dir (ptr)
48: char *ptr;
49: {
50:     int err;
51:
52:     err = func_39 (ptr);
53:     switch (err) {
54:     case 0x00:
55:         printf ("ディレクトリ '%s' を作成しました.¥n", ptr);
56:         return;
57:
58:     case 0x03:
59:         printf ("パス名 '%s' が無効です¥n", ptr);
60:         exit (2);
61:
62:     case 0x05:
63:         printf ("ディレクトリがいっぱい or 同名のファイルが存在します.¥n");
64:         exit (3);
65:     }
66: }
```

● mkd.c

リスト7-50 はプログラム mkd.exe の C ソース部分です．同リストにおいて，関数 main では最初にプログラム・タイトルを表示したのち，コマンド・ライン・パラメータの解析を行います．

パラメータの解析ではパラメータの個数を調べ，もしパス名(ディレクトリ名)が指定されていなければ，プログラム mkd.exe の使用方法を表示してエラー・ストップします．もし，パス名が与えられていれば，そのパス名へのポインタを引数として関数 mk_dir を用いてサブ・ディレクトリの作成を行います．

関数 mk_dir では，ポインタ ptr に渡されたパス名を関数(サブルーチン)func_39 に渡してサブ・ディレクトリの作成を行います．次に関数 func_39 から返されたエラー・コードを参照して，そのコードが“0”の場合は正常終了し，もし関数 func_39 でエラーが発生した場合はエラー・コードにしたがってエラー・メッセージの表示を行います．

● mkdsub.asm

リスト7-51 はプログラム mkd.exe のアセンブリ・ソース部分です．同リストにおいて，サブルーチン(関数)func_39 はファンクション 39H を用い，引数 arg1 で指定されたパス名でサブ・ディレクトリの作成を行います．

このとき，もしファンクション 39H でエラーが発生した場合は，そのエラー・コードを AX レジスタに返し，正常に終了した場合は AX レジスタに 0 を返します．

◆ 生成方法

プログラム mkd.exe は，次の手順で分割アセンブル/コンパイルして作成します．

```
masm /ML mkdsub;
cl -J -As mkd.c mkdsub
```

◆ 実行サンプル

リスト7-52 はサブ・ディレクトリ作成プログラム mkd.exe の実行例を示しています．

① dir コマンドを用いてカレント・ドライブのルート・ディレクトリを確認する．

② プログラム mkd.exe を用いてサブ・ディレクトリ ¥test を作成する．

③ 同様にプログラム mkd.exe を用いて，その下にさらにサブ・ディレクトリ test1 を作成する．

④ dir コマンドでルート・ディレクトリの確認を行う．

⑤ サブ・ディレクトリ ¥test が作成されている．

⑥ dir コマンドで，そのサブ・ディレクトリ ¥test の内容を確認する．

⑦ すると，指定したとおりにサブ・ディレクトリ ¥test¥test1 が作成されている．

⑧ さらに，dir コマンドでディレクトリ ¥test¥test1 の内容を確認する．

⑨ すると，そのディレクトリ ¥test¥test1 は空のディレクトリであり，新しく作成されたものであることが確認できる．

〔リスト7-51〕
プログラム
mkdsub.asm

```
 1: ;*****************************************************************
 2: ;
 3: ;    機    能 :   ディレクトリ作成サブルーチン
 4: ;    ファンクション :   39H(ディレクトリの作成)
 5: ;    生    成 :   masm /ML mkdsub;
 6: ;
 7: ;*****************************************************************
 8:                 .MODEL   SMALL, C
 9:                 .CODE
10: ;*****************************************************************
11: ;
12: ;      ルーチン名 :   func_39
13: ;      機    能 :   ディレクトリの作成
14: ;      func     :   39H(サブ・ディレクトリの作成)
15: ;      入    力 :   arg1 ･･･ パス名へのポインタ
16: ;      出    力 :   AX  ･･･ 00H: エラーなし
17: ;                           03H: パス名が無効
18: ;                           05H: ディレクトリが作成できない
19: ;
20: ;*****************************************************************
21: func_39         PROC      arg1:PTR
22:                 mov       dx, arg1
23:                 mov       ah, 39h              ;ファンクション 39H
24:                 int       21h
25:                 jc        err
26:                 xor       ax, ax
27: err:
28:                 ret
29: func_39         ENDP
30:                 END
```

⑩ 次に，プログラム mkd.exe に対してサブ・ディレクトリ ¥test2¥test を作成するように指定する．

⑪ ここで，⑤ で確認したようにディレクトリ ¥test2 は存在しないのでエラーが発生する．

サブ・ディレクトリを削除する

リスト7-53(rmd.c) と **リスト7-54**(rmdsub.asm) は，サブ・ディレクトリの削除を行うプログラム rmd.exe のソース・リストです．

● rmd.c

リスト7-53 はプログラム rmd.exe の C ソース部分です．同リストにおいて，関数 main では最初にプログラム・タイトルを表示したのち，コマンド・ライン・パラメータの解析を行います．

〔リスト7-52〕 プログラム mkd.exe の実行例

```
R>dir ¥ ⏎ … カレント・ドライブのルート・ディレクトリを確認 ①

 ドライブ R: のディスクのボリュームラベルは [IOS-10    ]
 ディレクトリは  R:¥

COMMAND  COM    24161   87-10-23    0:00
WK             <DIR>    88-12-07    8:01
      2 個のファイルがあります．
   636928 バイトが使用可能です．

R>mkd ¥test ⏎… サブ・ディレクトリ test を作る ②

 *** ディレクトリ作成プログラム  Ver.1.1 ***

ディレクトリ '¥test' を作成しました．

R>mkd ¥test¥test1 ⏎… その下にさらにサブ・ディレクトリ test1 を作る ③

*** ディレクトリ作成プログラム  Ver.1.1 ***

ディレクトリ '¥test¥test1' を作成しました．

R>dir ¥ ⏎… ルート・ディレクトリの確認 ④

 ドライブ R: のディスクのボリュームラベルは [IOS-10    ]
 ディレクトリは  R:¥

COMMAND  COM    24161   87-10-23    0:00
WK             <DIR>    88-12-07    8:01
TEST           <DIR>    88-12-07   12:18 ←(サブ・ディレクトリ test が作られている ⑤)
      3 個のファイルがあります．
   632832 バイトが使用可能です．

R>dir ¥test ⏎… サブ・ディレクトリ test を確認 ⑥

 ドライブ R: のディスクのボリュームラベルは [IOS-10    ]
 ディレクトリは  R:¥TEST

.              <DIR>    88-12-07   12:18
..             <DIR>    88-12-07   12:18
TEST1          <DIR>    88-12-07   12:18 ←(サブ・ディレクトリ test1 が作られている ⑦)
      3 個のファイルがあります．
   630784 バイトが使用可能です．

R>dir ¥test¥test1  ⏎…サブ・ディレクトリ ¥test¥test1 の確認 ⑧

 ドライブ R: のディスクのボリュームラベルは [IOS-10    ]
 ディレクトリは  R:¥TEST¥TEST1

.              <DIR>    88-12-07   12:18
..             <DIR>    88-12-07   12:18 ←(新しいディレクトリである ⑨)
      2 個のファイルがあります．
   630784 バイトが使用可能です．

R>mkd ¥test2¥test  ⏎… サブ・ディレクトリ ¥test2 ¥test の作成を指定 ⑩

 *** ディレクトリ作成プログラム  Ver.1.1 ***

パス名 '¥test2¥test' が無効です←(ルート・ディレクトリに test 2 が存在しないのでエラーになる ⑪)

R>
```

パラメータの解析ではパラメータの個数を調べ，もしパス名(ディレクトリ名)が指定されていなければ，プログラム rmd.exe の使用方法を表示してエラー・ストップします．もし，パス名が与えられていれば，そのパス名へのポインタを引数として関数 rm_dir を用いてサブ・ディレクトリの削除を行います．

関数 rm_dir では，ポインタ ptr に渡されたパス名を関数(サブルーチン)func_3a に渡してサブ・ディレクトリの削除を行います．次に関数 func_3a から返されたエラー・コードを参照して，そのコードが0の場

〔リスト7-53〕
プログラム rmd.c

```
 1: /*********************************************************************
 2: *
 3: *    機    能 :    ディレクトリの削除
 4: *    サ    ブ :    rmdsub.asm
 5: *    生    成 :    masm /ML rmdsub;
 6: *                  cl -J -AS rmd.c rmdsub
 7: *    使用方法 :    rmd <パス名>
 8: *
 9: **********************************************************************/
10: #include      <stdio.h>
11:
12: void main (int, char **);
13: void rm_dir (char *);
14: int  func_3a (char *);              /* サブ・ディレクトリの削除 */
15: /********************************************************************
16: *
17: *    関数名 :  main (argc, argv)
18: *    機  能 :  コマンドライン・パラメータの解析
19: *    入  力 :  int argc          ・・・・ コマンドライン・パラメータの数
20: *              char *argv[]      ・・・・ パラメータ文字列へのポインタ
21: *    出  力 :  なし
22: *
23: *********************************************************************/
24: void main (argc, argv)
25: int argc;
26: char **argv;
27: {
28:
29:     printf ("¥n *** ディレクトリ削除プログラム Ver.1.1 ***¥n¥n");
30:     if (argc != 2) {
31:         printf ("使用法 : ctrl コマンド名 引数 ・・・¥n");
32:         exit (1);
33:     }
34:     rm_dir (*++argv);
35:     printf ("¥n");
36:     exit (0);
37: }
38:
39: /********************************************************************
40: *
41: *    関数名 :  rm_dir (ptr)
42: *    機  能 :  ディレクトリの削除
43: *    入  力 :  ptr   パス名へのポインタ
44: *    出  力 :  なし
45: *
46: *********************************************************************/
47: void rm_dir (ptr)
48: char *ptr;
49: {
50:     int err;
51:
52:     err = func_3a (ptr);
53:     switch (err) {
54:     case 0x00:
55:         printf ("ディレクトリ '%s' を削除しました.¥n", ptr);
56:         return;
57:
58:     case 0x03:
59:         printf ("パス名 '%s' が無効です¥n" ptr);
60:         exit (2);
61:
62:     case 0x05:
63:         printf ("ディレクトリ '%s' にファイルが存在します.¥n", ptr);
64:         exit (3);
65:
66:     case 0x10:
67:         printf ("ルートまたはカレント・ディレクトリは削除できません.¥n");
68:         exit (4);
69:     }
70: }
```

〔リスト7-54〕プログラム rmdsub.asm

```
 1: ;*****************************************************************
 2: ;
 3: ;    機    能 :  ディレクトリ削除サブルーチン
 4: ;    ファンクション :  3AH（ディレクトリの削除）
 5: ;    生    成 :  masm /ML rmdsub;
 6: ;
 7: ;*****************************************************************
 8:             .MODEL  SMALL, C
 9:             .CODE
10: ;*****************************************************************
11: ;
12: ;    ルーチン名 :  func_3a
13: ;    機    能 :  ディレクトリの削除
14: ;    func    :  3AH（サブ・ディレクトリの削除）
15: ;    入    力 :  arg      パス名へのポインタ
16: ;    出    力 :  AX       00H: エラーなし
17: ;                         03H: パス名が無効
18: ;                         05H: ディレクトリが空でない
19: ;                         10H: ルート・ディレクトリは削除できない
20: ;
21: ;*****************************************************************
22: func_3a     PROC      arg1:PTR
23:             mov       dx, arg1
24:             mov       ah, 3Ah                     ;ファンクション 3AH
25:             int       21h
26:             jc        err
27:             xor       ax, ax
28: err:
29:             ret
30: func_3a     ENDP
31:             END
```

合は正常終了し，もし関数 func_3a でエラーが発生した場合はエラー・コードにしたがってエラー・メッセージの表示を行います．

● rmdsub.asm

リスト7-54 はプログラム rmd.exe のアセンブリ・ソース部分です．同リストにおいて，サブルーチン（関数）func_3a はファンクション 3AH を用い，引数 arg1 で指定されたパス名でサブ・ディレクトリの削除を行います．

このとき，もしファンクション 3AH でエラーが発生した場合は，そのエラー・コードを AX レジスタに返し，正常に終了した場合は AX レジスタに 0 を返します．

◆ **生成方法**

プログラム rmd.exe は，次の手順で分割アセンブル/コンパイルして削除します．

masm /ML rmdsub;

cl −J −As rmd.c rmdsub

◆ **実行サンプル**

リスト7-55 はサブ・ディレクトリ削除プログラム rmd.exe の実行例を示しています．

① dir コマンドを用いてカレント・ドライブのルート・ディレクトリを確認する．

② ここでサブ・ディレクトリ ¥test の存在に注目．

③ 次に，dir コマンドを用いてそのディレクトリ ¥test の確認を行う．

④ ここで，サブ・ディレクトリ test1 の存在に注目．

⑤ dir コマンドを用いてサブ・ディレクトリ ¥test¥test1 の確認を行う．

⑥ このディレクトリは空になっていて削除可能である．

⑦ プログラム rmd.exe を用いてサブ・ディレクトリ ¥test¥test1 の削除を行う．

⑧ 同様にプログラム rmd.exe を用いてサブ・ディレクトリ ¥test を削除する．

⑨ dir コマンドを用いてルート・ディレクトリの確認を行う．

⑩ サブ・ディレクトリ ¥test が削除されている．

⑪ ここで，プログラム rmd.exe に対して故意に無効なパス名を与えて起動する．

⑫ エラー・メッセージが表示される．

⑬ 次に，プログラム rmd.exe に対してルート・ディレクトリを削除するように指定する．

⑭ エラー・メッセージが表示される．

〔リスト7-55〕プログラム rmd.exe の実行例

```
R>dir ¥  ⏎… ルート・ディレクトリの確認①

 ドライブ R: のディスクのボリュームラベルは [IOS-10     ]
 ディレクトリは R:¥

COMMAND  COM     24161  87-10-23   0:00
WK           <DIR>      88-12-07   8:01
TEST         <DIR>      88-12-07  12:49 ←(サブ・ディレクトリ test の存在に注目②)
        3 個のファイルがあります.
    585728 バイトが使用可能です.

R>dir ¥test  ⏎… サブ・ディレクトリ ¥test の確認③

 ドライブ R: のディスクのボリュームラベルは [IOS-10     ]
 ディレクトリは R:¥TEST

.            <DIR>      88-12-07  12:49
..           <DIR>      88-12-07  12:49
TEST1        <DIR>      88-12-07  12:49 ←(サブ・ディレクトリ test1 の存在に注目④)
        3 個のファイルがあります.
    585728 バイトが使用可能です.

R>dir ¥test¥test1  ⏎… サブ・ディレクトリ ¥test¥test1 の確認⑤

 ドライブ R: のディスクのボリュームラベルは [IOS-10     ]
 ディレクトリは R:¥TEST¥TEST1

.            <DIR>      88-12-07  12:49 ←(空のディレクトリである⑥)
..           <DIR>      88-12-07  12:49
        2 個のファイルがあります.
    585728 バイトが使用可能です.

R>rmd ¥test¥test1  ⏎… サブ・ディレクトリ ¥test¥test1 を削除⑦

 *** ディレクトリ削除プログラム Ver.1.1 ***

ディレクトリ '¥test¥test1' を削除しました.

R>rmd ¥test  ⏎… サブ・ディレクトリ ¥test も削除する⑧

 *** ディレクトリ削除プログラム Ver.1.1 ***

ディレクトリ '¥test' を削除しました.

R>dir ¥  ⏎… ルート・ディレクトリを確認⑨

 ドライブ R: のディスクのボリュームラベルは [IOS-10     ]
 ディレクトリは R:¥

COMMAND  COM     24161  87-10-23   0:00 } サブ・ディレクトリ test が削除されている⑩
WK           <DIR>      88-12-07   8:01 }
        2 個のファイルがあります.
    589824 バイトが使用可能です.

R>rmd ¥test  ⏎… 存在しないディレクトリ名を引数として与える⑪

 *** ディレクトリ削除プログラム Ver.1.1 ***

パス名 '¥test' が無効です ←(エラー・メッセージ⑫)

R>rmd ¥  ⏎… ルート・ディレクトリの削除を指定⑬

 *** ディレクトリ削除プログラム Ver.1.1 ***

ルートまたはカレント・ディレクトリは削除できません. ←(エラー・メッセージ⑭)

R>
```

カレント・ディレクトリの表示/変更を行う

リスト7-56(chd.c)および**リスト7-57**(chdsub.asm)は，カレント・ドライブの変更や表示を行うためのプログラムchd.exeのソース・リストです．

● chd.c

リスト7-56はプログラムchd.exeのCソース部分です．同リストにおいて，関数mainでは，最初にプログラム・タイトルを表示したのち，コマンド・ライン・パラメータの解析を行います．

パラメータの解析では，まずパラメータの個数を調べ，パラメータがなければ関数(サブルーチン)func_19を用いてカレント・ドライブ番号の読み出しを行い，そのドライブのカレント・ディレクトリを関数disp_dirによって表示します．

パラメータでドライブのみが指定されている場合は，その指定されたドライブのカレント・ディレクトリを関数dsp_dirによって表示します．もし，パラメータにパス名が指定されている場合には，指定されたドライブ(カレント・ドライブも含む)のカレント・ディレクトリを，指定されたディレクトリに関数chg_dirを用いて変更します．

関数dsp_dirは，引数として渡されたドライブ番号を関数(サブルーチン)func_47に渡してカレント・ディレクトリの読み出しを行います．

関数func_47では，読み出しが正常に行われた場合には，指定された文字列バッファbuffにカレント・ディレクトリ名を格納して返します．もし，カレント・ディレクトリの読み出しに失敗すると，関数func_47から0以外(エラー・コード)が返されるので，その旨を表示してエラー・ストップします．

〔リスト7-56〕 プログラムchd.c ①

```
 1: /*********************************************************************
 2: *
 3: *   機　　能 :  カレント・ディレクトリの変更／表示
 4: *   サ　　ブ :  chdsub.asm
 5: *   生　　成 :  masm /ML chdsub;
 6: *               cl -J -AS chd.c chdsub
 7: *   使用方法 :  chd [<パス名>]
 8: *
 9: *********************************************************************/
10: #include     <stdio.h>
11: #include     <string.h>
12:
13: void main (int, char **);
14: void dsp_dir (int);
15: void chg_dir (char *);
16: int  func_19 (void);              /* カレント・ドライブ番号の読み出し */
17: int  func_3b (char *);            /* カレント・ディレクトリの変更   */
18: int  func_47 (int, char *);       /* カレント・ディレクトリの読み出し */
19: /*********************************************************************
20: *
21: *   関数名 :  main (argc, argv)
22: *   機　能 :  コマンドライン・パラメータの解析
23: *   入　力 :  int argc       ‥‥‥  コマンドライン・パラメータの数
24: *             char *argv[]   ‥‥‥  パラメータ文字列へのポインタ
25: *   出　力 :  なし
26: *
27: *********************************************************************/
28: void main (argc, argv)
29: int argc;
30: char **argv;
31: {
32:     int drv_num = 0;
33:
34:     printf ("¥n *** カレント・ディレクトリ変更／表示プログラム Ver.1.1 ***¥n¥n")
35:     if (argc == 1) {
36:         drv_num = func_19 ();
37:         dsp_dir (drv_num);
38:     } else {
39:         if (strchr (*++argv + 1, ':') != NULL) {
40:             drv_num = toupper (**argv) - 'A';
41:             if (strlen (*argv) > 2) {
42:                 chg_dir (*argv);
43:             } else {
44:                 dsp_dir (drv_num);
45:             }
46:         } else {
47:             chg_dir (*argv);
48:         }
49: }
```

関数func_47でカレント・ディレクトリの読み出しに成功したら，ドライブ名とカレント・ディレクトリの表示を行います．

関数chg_dirは，ポインタptrで渡されたパス名を関数func_3bに渡し，カレント・ディレクトリの変更を行います．ここでも，関数func_3bで正常に処理(ディレクトリ変更)が行われれば0が返され，エラーが発生すればエラー・コードが返されるのでそれぞれの状態を表示します．

● chdsub.asm

リスト7-57はプログラムchd.exeのアセンブリ・ソース部分です．同リストにおいて，サブルーチン(関数)func_19は，ファンクション19Hを用いてカレント・ドライブ番号の読み出しを行い，そのドライブ番号をAXレジスタに返します．

サブルーチンfunc_3dは，ファンクション3BHを用いてポインタarg1で指定されたパス名(ディレクトリ名)にカレント・ディレクトリの変更を行います．ここで，ファンクション3BHでエラーが発生すればAXレジスタにそのエラー・コードを，正常に終了すればAXレジスタに0を返します．

サブルーチンfunc_47は，ファンクション47Hを用いてarg1で指定されたドライブのカレント・ディレクトリを読み出し，arg2で指定されたバッファに格納します．ここでも，ファンクションが正常に終了したらAXレジスタには0を，エラーが発生したらそのエラー・コードをAXレジスタに返します．

◆ **生成方法**

プログラムchd.exeは，下記の手順で分割アセンブル/コンパイルして作成します．

〔リスト7-56〕プログラムchd.c ②

```
50:     printf ("¥n");
51:     exit (0);
52: }
53:
54: /***************************************************************
55: *
56: *     関数名：   dsp_dir (drv_num)
57: *     機　能：   カレント・ディレクトリの表示
58: *     入　力：   drv_num    ドライブ番号
59: *     出　力：   なし
60: *
61: ****************************************************************/
62: void dsp_dir (drv_num)
63: int drv_num;
64: {
65:     char buff[64];
66:
67:     if (func_47 (drv_num + 1, buff)) {
68:         printf ("ドライブの指定が無効です");
69:         exit (1);
70:     }
71:     printf ("ドライブ %c のカレント・ディレクトリは", drv_num + 'A');
72:     printf (" '¥¥%s' です.¥n", buff);
73: }
74:
75: /***************************************************************
76: *
77: *     関数名：   chg_dir (ptr)
78: *     機　能：   カレント・ディレクトリの変更／表示
79: *     入　力：   ptr   ：パス名へのポインタ
80: *     出　力：   なし
81: *
82: ****************************************************************/
83: void chg_dir (ptr)
84: char *ptr;
85: {
86:     int err;
87:
88:     err = func_3b (ptr);
89:     switch (err) {
90:     case 0x00:
91:         printf ("カレント・ディレクトリを '%s' に変更しました.¥n", ptr);
92:         return;
93:
94:     case 0x03:
95:         printf ("パス名 '%s' が無効です¥n", ptr);
96:         exit (2);
97:     }
98: }
```

masm /ML chdsub;

cl −J −As chd.c chdsub

◆ **実行サンプル**

リスト7-58 はカレント・ディレクトリの表示/変更を行うプログラムchd.exeの実行例を示しています．

① まず，dirコマンドを用いてドライブH(ハード・ディスク)のカレント・ディレクトリの確認を行う．

② ここで，ドライブHのカレント・ディレクトリが¥wk1¥wkであることに注目．

③ 次に，プログラムchd.exeに対してパラメータと

〔リスト7-57〕
プログラム
chdsub.asm

```
 1: ;***********************************************************************
 2: ;
 3: ;   機      能   カレント・ディレクトリ変更サブルーチン
 4: ;   ファンクション  19H（カレント・ドライブ番号の読み出し）
 5: ;                3BH（カレント・ディレクトリの変更）
 6: ;                47H（カレント・ディレクトリの読み出し）
 7: ;   生      成   masm /ML chdsub;
 8: ;
 9: ;***********************************************************************
10:             .MODEL  SMALL, C
11:             .CODE
12: ;***********************************************************************
13: ;
14: ;       ルーチン名 :   func_19
15: ;       機    能 :   カレント・ドライブ番号の取得
16: ;       func     :   19H（カレント・ドライブ番号の読み出し）
17: ;       入    力 :   なし
18: ;       出    力 :   AX ･･･ ドライブ番号（00H=A, 01H=B, ･･･）
19: ;
20: ;***********************************************************************
21: func_19     PROC
22:             mov     ah, 19h                 ;ファンクション19H
23:             int     21h
24:             xor     ah, ah
25:             ret
26: func_19     ENDP
27:
28: ;***********************************************************************
29: ;
30: ;       ルーチン名 :   func_3b
31: ;       機    能 :   カレント・ディレクトリの変更
32: ;       func     :   3BH（サブ・カレント・ディレクトリの変更）
33: ;       入    力 :   arg ･･･ パス名へのポインタ
34: ;       出    力 :   AX  ･･･ 00H: エラーなし
35: ;                            03H: パス名が無効
36: ;
37: ;***********************************************************************
38: func_3b     PROC    arg1:PTR
39:             mov     dx, arg1
40:             mov     ah, 3Bh                 ;ファンクション3BH
41:             int     21h
42:             jc      err
43:             xor     ax, ax
44: err:
45:             ret
46: func_3b     ENDP
47:
48: ;***********************************************************************
49: ;
50: ;       ルーチン名 :   func_47
51: ;       機    能 :   カレント・ディレクトリの読み出し
52: ;       func     :   47H（サブ・カレント・ディレクトリの読み出し）
53: ;       入    力 :   arg1 ･･･ ドライブ番号
54: ;                    arg2 ･･･ バッファへのポインタ
55: ;       出    力 :   AX       00H: エラーなし
56: ;                             03H: ドライブ名が無効
57: ;
58: ;***********************************************************************
59: func_47     PROC    arg1:WORD, arg2:PTR
60:             push    si
61:             mov     dx, arg1
62:             mov     si, arg2
63:             mov     ah, 47h                 ;ファンクション47H
64:             int     21h
65:             jc      err
66:             xor     ax, ax
67: err:
68:             pop     si
69:             ret
70: func_47     ENDP
71:             END
```

してドライブ名(H:)を指定する.

④ これによって，指定されたドライブ(H:)のカレント・ディレクトリが読み出されて表示される.

⑤ 次にプログラム chd.exe に対しドライブ名の付いたパス名(ルート・ディレクトリ)を指定する.

⑥ これによってドライブ H のカレント・ディレクトリがルート・ディレクトリに変更される.

⑦ dir コマンドを用いてドライブ H のカレント・ディレクトリを確認する.

⑧ 指定されたとおりにルート・ディレクトリに変更されている.

⑨ dir コマンドを用いてカレント・ディレクトリ(ドライブ R)のカレント・ディレクトリを確認する.

⑩ ドライブ R のカレント・ディレクトリは ¥wk であることに注目.

⑪ プログラム chd.exe をパラメータなしで起動する. これによって，カレント・ドライブのカレント・ディレクトリが読み出される.

⑫ カレント・ドライブのカレント・ディレクトリは ¥wk であり，dir コマンドで確認したのと一致する.

⑬ 次に，プログラム chd.exe に対してパラメータとしてディレクトリ名 test を与えて起動する.

⑭ 指定されたとおりにカレント・ドライブのカレント・ディレクトリは ¥wk¥test に変更されているはずである.

⑮ dir コマンドを用いてカレント・ドライブのカレント・ディレクトリを確認する.

⑯ 指定どおりにカレント・ディレクトリが ¥wk ¥test に変更されている.

〔リスト7-58〕
プログラム
chd.exe の実行例 ①

```
R>dir h: ⏎… ドライブHのカレント・ディレクトリを確認①

 ドライブ H: のディスクのボリュームラベルは HD1
 ディレクトリは H:¥WK1¥WK ←(カレント・ディレクトリ②)

.               <DIR>       88-11-08  15:16
..              <DIR>       88-11-08  15:16
FATALSUB ASM     4781       88-12-07   8:36
CTRL     C       1364       88-12-07   9:04
TEE      C        402       88-11-28  15:02



GSTR     EXE    10026       88-12-12  11:25
DIV0     EXE      816       88-12-12  11:34
MALOC    EXE    12625       88-12-12  12:04
        141 個のファイルがあります.
    5750784 バイトが使用可能です.

R>chd h: ⏎… パラメータにドライブHを指定③

 *** カレント・ディレクトリ変更／表示プログラム Ver.1.1 ***

 ドライブ H のカレント・ディレクトリは '¥WK1¥WK' です.
                                  (カレント・ディレクトリ④)

R>chd h:¥ ⏎… パス名(ルート・ディレクトリ)の指定⑤

 *** カレント・ディレクトリ変更／表示プログラム Ver.1.1 ***

 カレント・ディレクトリを 'h:¥' に変更しました.
                  (ルート・ディレクトリに変更⑥)

R>dir h: ⏎… ドライブHのカレント・ディレクトリを確認⑦

 ドライブ H: のディスクのボリュームラベルは HD1
 ディレクトリは H:¥ ←(指定どおりに変更されている⑧)

COM1            <DIR>       88-11-05   9:43
COM3            <DIR>       88-11-05   9:43
SYS             <DIR>       88-11-05   9:44



WK1             <DIR>       88-11-05   9:45
WK2             <DIR>       88-11-05   9:45
WK3             <DIR>       88-11-05   9:45
```

〔リスト7-58〕
プログラム
chd.exe の実行例 ②

```
        24 個 の フ ァ イ ル が あ り ま す .
  5750784 バ イ ト が 使 用 可 能 で す .

R>dir ⏎… カレント・ドライブのカレント・ディレクトリの確認 ⑨

 ド ラ イ ブ  R: の デ ィ ス ク の ボ リ ュ ー ム ラ ベ ル は  [IOS-10     ]
 デ ィ レ ク ト リ は  R:¥WK ← (カレント・ディレクトリ ⑩)
.            <DIR>      88-12-12   8:18
..           <DIR>      88-12-12   8:18
CHDSUB   ASM     2183   88-12-12  13:03
SUB      ASM     1242   88-12-07   8:39
DIRCOM   ASM     1084   85-12-09  14:25
CTRL     C       1366   88-12-12   9:16
FATAL    C       3620   88-12-12  10:07
UPPER    C        769   88-12-12  10:40
       156 個 の フ ァ イ ル が あ り ま す .
  393216 バ イ ト が 使 用 可 能 で す .

R>chd ⏎… パラメータなしで起動 ⑪

 *** カ レ ン ト ・ デ ィ レ ク ト リ 変 更 / 表 示 プ ロ グ ラ ム  Ver.1.1 ***

ド ラ イ ブ  R の カ レ ン ト ・ デ ィ レ ク ト リ は  '¥WK'  で す .
            ↑(カレント・ドライブ)                ↑(カレント・ディレクトリ ⑫)

R>chd test ⏎… ディレクトリ名の指定 ⑬

 *** カ レ ン ト ・ デ ィ レ ク ト リ 変 更 / 表 示 プ ロ グ ラ ム  Ver.1.1 ***

カ レ ン ト ・ デ ィ レ ク ト リ を  'test' に 変 更 し ま し た .
                                   ↑(カレント・ディレクトリの変更 ⑭)
R>dir ⏎… カレント・ディレクトリの確認 ⑮

 ド ラ イ ブ  R: の デ ィ ス ク の ボ リ ュ ー ム ラ ベ ル は  [IOS-10     ]
 デ ィ レ ク ト リ は  R:¥WK¥TEST ← (指定どおりに変更されている ⑯)

.            <DIR>      88-12-12  13:06
..           <DIR>      88-12-12  13:06
         2 個 の フ ァ イ ル が あ り ま す .
  393216 バ イ ト が 使 用 可 能 で す .

R>
```

階層ディレクトリを操作する

リスト7-59(d.c)および**リスト7-60**(dsub.asm)は，階層ディレクトリを表示したり，ディレクトリのソート(並べ替え)を行って表示するディレクトリ表示プログラムd.exeのソース・リストです．

● d.c

リスト7-59はプログラムd.exeのCソース部分です．同リストにおいて，構造体 _DTA は，ファイル検索に使用されるディスク転送バッファ(DTA)の構造を定義しています．

構造体 _DIR は，ディレクトリのソートを行う際にファイル情報を格納しておくためのデータ領域の構造を定義しています．また，ファイル属性を表す各種フラグやファイルの数を計数するための変数はグローバル変数として宣言して，どの関数からも自由にアクセス可能としています．

関数mainでは，最初にプログラム・タイトルを表示したのち，各種フラグやポインタなどの初期化を行っておきます．次に，コマンド・ライン・パラメータで指定されたオプションの解析を行います．ここで，プログラムd.exeの起動オプションは**表7-1**のようになっていて，表示するファイルの属性を指定したり，ソートの有無やファイル数表示の有無などの指定を行います．オプションの解析では，それぞれのオプションに対応したフラグを設定しておき，実際のオプションに対応した処理は各関数内で行います．

オプションの解析が終わったら，次に関数path_setを用いてパス名の解析を行い，その解析結果と起動オプションの指定にしたがってディレクトリの表示を行います．すべてのディレクトリ表示が終了したら，

関数af_dispを用いてディレクトリ数やファイル数の合計の数を表示してプログラムd.exeを終了します．

関数path_setでは，ポインタpathで指定されたパス名の解析を行い，パス名をドライブ名drv，サブ・ディレクトリ名subdir，ファイル名fileにそれぞれ分けて格納します．まず，ドライブ名やサブ・ディレクトリ名，ファイル名の各バッファの初期化を行ったのち，関数(サブルーチン)func_1aを用いてDTAの設定を行います．

そして，もしドライブ名が指定されていれば，そのドライブ名をバッファdrvに格納します．もし，ドライブ名が指定されていなければ，関数func_19を用いてカレント・ドライブ番号を読み出し，そのドライブ番号をドライブ名に変換してバッファdrvに格納します．次に，関数vol_msgにドライブ名を渡して，そのドライブ名とメディアのボリューム・ラベル名の表示を行います．

フル・パス名(ルート・ディレクトリから始まるパス名)が省略されている場合は，関数func_47を用いて，指定されたドライブのカレント・ディレクトリを読み出して，バッファsubdirに格納します．

もしパス名の中にサブ・ディレクトリ名が指定されている場合は，そのサブ・ディレクトリ名を切り出してバッファsubdirに格納しておきます．

これらの処理が終わったら，再びドライブ名とサブ・ディレクトリ名，ファイル名を連結してパス名とし，そのパス名を関数func_4eに渡してファイル検索を行います．ここで，ファイル検索の結果，そのパス名がファイル名ではなくサブ・ディレクトリ名の場合は，そのサブ・ディレクトリ(パス名)名をバッファsubdirに連結します．もし，サブ・ディレクトリ名でなければ，ファイル名としてバッファfileに格納しておきます．

これらの処理でパス名をドライブ名，サブ・ディレクトリ名，ファイル名に分解できたのでパスの表示を行ったのち，これらの名前を連結してパス名として関数dirに渡してディレクトリの表示を行います．

関数dirは，リカーシブ(再帰呼び出し可能)な関数で，ポインタptrで指定されたパス名(ファイル名またはサブ・ディレクトリ名)の表示を行います．この際に，引数levelで指定されたディレクトリ・レベルに対応して段下げを行って，階層ディレクトリの表示を見やすくしています．

また，プログラムd.exeではディレクトリを表示する際に，dirコマンドと異なり先にサブ・ディレクトリ名の表示を行い，そのあとにファイル名の表示を行うことにします．このため，関数dirでは最初に関数search_dirを用いてサブ・ディレクトリの検索を行い，もしサブ・ディレクトリが存在すれば，さらにその下のディレクトリの表示を行っています．

関数dirは再帰的に呼ばれるため，関数func_2fを用いて現在のDTAをポインタdta_ptrに格納しておき，そののちに関数func_1aを用いて自動変数(スタック)に確保されたDTAの設定を行います．

次に，起動時オプションで指定された各種フラグから，読み出そうとするファイルの属性を変数atrに設定し，関数func_4eを用いてファイルの検索を行います．ここで，もし指定されたファイルが存在しない場合は，関数func_1aによってDTAの復帰を行い，－mオプション(ファイル数表示の抑止)の指定がない場合は，関数f_dispを用いてファイル数の表示を行ってから関数dirをリターンします．

もし，指定されたファイルが存在したら，do～whileループと関数func_4fを用いて，指定された属性をもつファイルの数を調べます．ここで，指定された属性のファイルが存在しない場合は，関数func_1aによってDTAの復帰を行って関数dirをリターンします．

もし，指定された属性のファイルが存在したら，ファイル数の合計を計算したのち，ライブラリ関数callocを用いてファイル情報を格納するためのデータ領域を確保します．ここで，もしメモリ不足によってデータ領域の確保ができない場合はエラー・メッセージを表示してプログラムd.exeを終了します．

データ領域が確保されたら，関数func_4eを用いて再びファイルの検索を行います．そして，関数func_4fとdo～whileループを用いてDTAに得られたファイル名やサイズ，更新日時などのファイルに関する情報を，一致したファイルがなくなるまで関数callocによって確保したデータ領域にコピーしていきます．

これで，あるディレクトリ内における指定されたファイルの情報が，関数callocによって確保されたデータ領域に揃ったことになるので，－qオプション(ソート)が指定されている場合には，MS-Cのライブラリ関数である関数qsortを用いて，得られたファイル情報の並べ替えを行います．ここで，関数qsortでは，並べ替えの際に必要となる比較のための関数は，ユーザが用意することによって柔軟に対応できるようになっています．プログラムd.exeの場合は，関数dir_compがファイル情報の比較を担当するので，その関数アドレスを関数qsortに渡しています．

関数qsortによってファイル情報の並べ替えが終わったら，do～whileループを用いてファイル情報を関数disp_dirに渡してファイル情報の表示を行います．

do～whileループによって，すべてのファイル情報の表示が終わったら，関数callocによって確保したデ

ータ領域を関数freeを用いて返却します．このあと，関数func_1aを用いてDTAの復帰を行って関数dirからリターンします．

関数search_dirもリカーシブな関数です．この関数も，関数func_2fを用いて現在のDTAをポインタdta_ptrに格納しておき，そののちに関数func_1aを用いてスタック上のDTAを設定します．

さて，起動時に−dオプション(ディレクトリの再帰表示)が指定されていれば，その下のディレクトリ内容を調べるため，パス名に“*.*”を連結しておきます．そして，関数func_4eを用いて指定ディレクトリの検索を行います．ここで，もし指定されたディレクトリが存在しない場合は，関数func_1aによってDTAの復帰を行ってから関数search_dirをリターンします．

もし，指定されたディレクトリ内にファイルが存在したら，do〜whileループと関数func_4fを用いて，そのディレクトリ内のファイル属性を調べます．ここで，そのファイルがサブ・ディレクトリ(“.”や“..”は除く)の場合だけサブ・ディレクトリの数をカウントします．

また，起動時に−mオプション(ディレクトリ数表示の抑止)が指定されていなければ，関数d_dispを用いてサブ・ディレクトリの数を表示します．

サブ・ディレクトリの数が0の場合は，関数func_1aによってDTAの復帰を行って関数search_dirをリターンします．もし，サブ・ディレクトリの数が0でなければ，グローバル変数のサブ・ディレクトリの合計数を更新し，ライブラリ関数callocを用いてディレクトリ情報を格納するためのデータ領域を確保します．ここで，もしメモリ不足によってデータ領域の確保ができない場合は，エラー・メッセージを表示してプログラムd.exeを終了します．

データ領域が確保されたら，再び関数func_4eおよび関数func_4fとdo〜whileループによって，DTAに返されたディレクトリ・エントリの情報から必要な情報を，関数callocによって確保したデータ領域にコピーしていきます．

ディレクトリ情報の収集が終わったら，−qオプションの有無を調べ，もし−qオプション(ソート)が指定されている場合には，MS-Cのライブラリ関数である関数qsortを用いて，得られたディレクトリ情報の並べ替えを行います．関数qsortによってディレクトリ情報の並べ替えが終わったら，do〜whileループを用いてディレクトリ情報を関数disp_dirに渡して，サブ・ディレクトリ名の表示を行います．

起動時に−dオプション(ディレクトリの階層的表示)が指定されていれば，関数dirを再帰的に呼び出して，その下のディレクトリ内容の表示を繰り返します．

すべてのディレクトリ情報の表示が終わったら，関数callocによって確保したデータ領域を関数freeを用いて返却します．このあと，関数func_1aを用いてDTAの復帰を行って関数serach_dirをリターンします．

関数vol_msgは，ポインタdrvで指定されたドライブのボリューム・ラベルを表示します．この関数では，まず関数func_2fを用いてDTAの読み出しを行い，そのアドレスをポインタdta_ptrに格納しておきます．次に，ドライブ名へのポインタdrvを用いてドライブ名の表示を行ったのち，関数func_4eによって指定されたドライブのルート・ディレクトリを検索し，ボリューム・ラベルを探して表示します．これらの処理が終わったら，関数func_1aを用いてDTAの復帰を行って関数vol_msgからリターンします．

関数disp_dirは，ファイル名やディレクトリ名などディレクトリ・エントリの情報を表示します．この関数では，まず最初に関数atr_dispを用いて属性フィールドの表示を行い，次にDTAに返されるパックされたファイル名を関数name_setを用いて12文字フォーマットのファイル名に変換して表示します．

次にファイル・サイズの表示を行いますが，ファイルの属性がサブ・ディレクトリの場合には，サイズが0なのでサイズ・フィールドに〈DIR〉を表示します．これらの表示が終わったらファイルの最終更新日時を表示して関数disp_dirからリターンします．

関数atr_dispは，属性フィールドの表示を行います．この関数では，まず引数levelで指定されたディレクトリ・レベルに対応して段下げを行い，次に引数atrで指定された属性に対応する各ビット位置に属性の頭文字を表示していきます．

関数name_setは，DTAから返されたファイル名(パックされた名前)を12文字のフィールドに“名前.拡張子”のフォーマットに変換して格納します．

関数f_dispはファイルの数を表示します．この際に，引数levelで指定されたディレクトリ・レベルに対応して段下げを行います．

関数d_dispはサブ・ディレクトリの数を表示します．この関数でも，引数levelで指定されたディレクトリ・レベルに対応して段下げを行います．

関数af_dispは，サブ・ディレクトリの数とファイルの数の合計の数を表示します．

関数dir_compは，ライブラリ関数qsortから呼び出され，二つの要素(ファイル情報)へのポインタを引数として受け取り，それらの要素の比較を行います．比較の結果，もし二つの要素が同じ場合には“0”を返

し，第一の要素(x)が第二の要素(y)よりも大きい場合は正の値を，小さい場合は負の値を返します．ここで，要素としてファイル情報へのポインタが渡されてくるので，もし起動時に－qtオプション(日時によるソート)が指定されている場合には，日付，時刻の順で比較を行い，その結果を“1,0,－1”のいずれかの値で返します．

もし，－qtオプションでない場合は－qfオプション(名前によるソート)なので，ライブラリ関数strcompによってファイル名の比較を行い，その結果を戻り値として返しています．

〔表7-1〕プログラムd.exeの起動オプション

オプション	機　　　能
－d	サブ・ディレクトリの内容も表示
－h	システム・ファイルやシークレット・ファイルも表示
m	ファイル数表示の抑止
－q〔f〕	ファイル名によるソート
－qt	タイム・スタンプによるソート
－r	読み出し専用ファイルの表示

〔リスト7-59〕プログラムd.c ①

```
 1: /*****************************************************************
 2: *
 3: *   機    能 : ディレクトリの表示
 4: *   サ    ブ : dsub.asm
 5: *   生    成 : masm /ML dsub;
 6: *              cl -J -AS d.c dsub -link /STACK:4096
 7: *   使用方法 : d <パス名>
 8: *
 9: ******************************************************************/
10: #define     NAME_SIZE   8
11: #define     EXT_SIZE    3
12: #define     NAME_LEN    NAME_SIZE + EXT_SIZE + 2
13: #define     PATH_LEN    64
14: #define     DRV_LEN     3
15: #define     ATR_READ    0x01
16: #define     ATR_BRND    0x02
17: #define     ATR_SYS     0x04
18: #define     ATR_VOL     0x08
19: #define     ATR_DIR     0x10
20: #define     ATR_ARC     0x20
21:
22: #include    <stdio.h>
23: #include    <string.h>
24: #include    <malloc.h>
25: #include    <memory.h>
26: #include    <search.h>
27: /*****************************************************************
28: *
29: *    構造体 :   _DTA
30: *    機  能 :   DTAのデータ構造の定義
31: *
32: ******************************************************************/
33: typedef struct _DTA {
34:     char atr0;              /*00H         検索するファイルの属性 */
35:     char drv;               /*01H         ドライブ番号(00H=A, 01H=B) */
36:     char file [NAME_SIZE];  /*02H - 09H パス名のファイル名部分 */
37:     char ext [EXT_SIZE];    /*0AH - 0CH パス名の拡張子部分 */
38:     char reserve [8];       /*0DH - 14H システム予約 */
39:     char atr;               /*15H         検索されたファイルの属性 */
40:     unsigned int  time;     /*16H - 17H 最終変更時刻 */
41:     unsigned int  date;     /*18H - 19H 最終変更日付 */
42:     long size;              /*1AH - 1DH ファイルの大きさ */
43:     char name [NAME_LEN];   /*1EH - 2AH パックされたファイル名 */
44: } DTA;
45:
46: /*****************************************************************
47: *
48: *    構造体 :   _DIR
49: *    機  能 :   ディレクトリ・バッファの構造定義
50: *
51: ******************************************************************/
52: typedef struct _DIR {
53:     char name [NAME_LEN];       /* パックされたファイル名 */
54:     int  atr;                   /* ファイルの属性 */
55:     unsigned int  time;         /* 最終変更時刻 */
56:     unsigned int  date;         /* 最終変更日付 */
57:     long size;                  /* ファイルの大きさ */
58: } DIR;
59:
```

〔リスト7-59〕プログラム d.c ②

```
 60: void main (int, char **);
 61: void path_set (char *);
 62: void vol_msg (char *);
 63: void dir (char *, int);
 64: void search_dir (char *, int);
 65: void disp_dir (char *, int, long, unsigned int, unsigned int, int);
 66: void atr_disp (int, int);
 67: void name_set (char *, char *);
 68: void f_disp (int, int);
 69: void d_disp (int, int);
 70: void af_disp (void);
 71: int  dir_comp (DIR *, DIR *);
 72: int  func_19 (void);                   /* カレント・ドライブ番号の読み出し */
 73: void func_1a (DTA *);                  /* ＤＴＡの設定 */
 74: DTA *func_2f (void);                   /* ＤＴＡの読み出し */
 75: void func_47 (int, char *);            /* カレント・ディレクトリの読み出し */
 76: int  func_4e (char *, int);            /* 一致するファイルの検索 */
 77: int  func_4f (void);                   /* つぎに一致するファイルの検索 */
 78:
 79: int R_flag, B_flag, S_flag, D_flag, A_flag, QF_flag, QT_flag, M_flag;
 80: int ad_count, af_count;
 81: /*************************************************************************
 82: *
 83: *    関数名：  main (argc, argv)
 84: *    機  能：  コマンドライン・パラメータの解析
 85: *    入  力：  int argc           コマンドライン・パラメータの数
 86: *              char *argv[] ・・・・ パラメータ文字列へのポインタ
 87: *    出  力：  なし
 88: *
 89: *************************************************************************/
 90: void main (argc, argv)
 91: int argc;
 92: char **argv;
 93: {
 94:         char *path;
 95:
 96:         printf ("¥n *** ディレクトリ表示プログラム Ver.1.1 ***¥n¥n");
 97:         path = "*.*";
 98:         R_flag = B_flag = S_flag = D_flag = 0;
 99:         M_flag = QT_flag = QF_flag = 0;
100:         af_count = ad_count = 0;
101:         A_flag = ATR_ARC;
102:         while (--argc > 0) {
103:                 if (***++argv == '-') {
104:                         if (toupper (argv[0][1])) {
105:                                 argv[0][1] = tolower (argv[0][1]);
106:                         }
107:                         switch (argv[0][1]) {
108:                         case 'd' :
109:                                 D_flag = ATR_DIR; break;
110:                         case 'h' :
111:                                 R_flag = ATR_READ;
112:                                 B_flag = ATR_BRND;
113:                                 S_flag = ATR_SYS; break;
114:                         case 'm' :
115:                                 M_flag = 1; break;
116:                         case 'q' :
117:                                 QF_flag = 1;
118:                                 if (tolower (argv[0][2]) == 't') {
119:                                         QT_flag = 1;
120:                                 }
121:                                 break;
122:                         case 'r' :
123:                                 R_flag = ATR_READ; break;
124:                         default:
125:                                 fprintf (stderr, "option error :¥"%c¥"¥n", argv[0][1]);
126:                                 break;
127:                         }
128:                 } else {
129:                         path = *argv;
130:                 }
131:         }
132:         path_set (path);
133:         af_disp ();
134:         exit (0);
135: }
136:
```

〔リスト7-59〕プログラムd.c ③

```
137: /**************************************************************
138: *
139: *    関数名 :  path_set (ptr)
140: *    機　能 :  ディレクトリの表示
141: *    入　力 :  ptr ･･･ パス名へのポインタ
142: *    出　力 :  なし
143: *
144: *************************************************************/
145: void path_set (path)
146: char *path;
147: {
148:     DTA dta;
149:     char buff [PATH_LEN];
150:     char drv [DRV_LEN];
151:     char subdir [PATH_LEN];
152:     char file [NAME_LEN];
153:     char *path_ptr;
154:     char *ptr;
155:     int c;
156:
157:     strcpy (drv, "A:");
158:     strcpy (subdir, "¥¥");
159:     strcpy (file, "*.*");
160:     func_1a (&dta);
161:     if (path_ptr = strchr (path, ':')) {
162:         *drv = toupper (*path);              /* ドライブ名セット */
163:         *path_ptr++ = '¥0';
164:     } else {
165:         *drv = func_19 () + 'A';
166:         path_ptr = path;                     /* デフォルト・ドライブ名 */
167:     }
168:     vol_msg (drv);                           /* ボリューム名表示 */
169:     if (*path_ptr != '¥¥') {
170:         func_47 (*drv - 'A' + 1, subdir + 1);  /* カレント・ディレクトリ */
171:         strcat (subdir, "¥¥");
172:     }
173:     c = *path_ptr;
174:     if (ptr = strrchr (path_ptr, '¥¥')) {
175:         *ptr++ = '¥0';
176:         if (*path_ptr == '¥¥') {
177:             *path_ptr++ = '¥0';
178:         }
179:         strcat (subdir, path_ptr);           /* ディレクトリ名の切り出し */
180:         if (*path_ptr != '¥0') {
181:             strcat (subdir, "¥¥");
182:         }
183:     } else {
184:         ptr = path_ptr;
185:     }
186:     if (! (strrchr (ptr, '*') || *ptr == '¥0' || *ptr == '¥¥')) {
187:         strcpy (buff, drv);
188:         strcat (buff, subdir);
189:         strcat (buff, ptr);
190:         if (! func_4e (buff, ATR_DIR) && dta.atr == ATR_DIR) {
191:             strcat (subdir, ptr);            /* ディレクトリ設定 */
192:             strcat (subdir, "¥¥");
193:         } else {
194:             strcpy (file, ptr);              /* ファイル設定 */
195:         }
196:     } else {
197:         if (*ptr != '¥0') {
198:             strcpy (file, ptr);
199:         }
200:     }
201:     printf ("%c:%s  ･･･  ディレクトリ¥n", *drv, subdir);
202:     strcpy (buff, drv);
203:     strcat (buff, subdir);
204:     strcat (buff, file);
205:     dir (buff, 1);
206: }
207:
208: /**************************************************************
209: *
210: *    関数名 :  dir (ptr, level)
211: *    機　能 :  ディレクトリ(ファイル名)の読み込み
212: *    入　力 :  ptr    ･･･ パス名へのポインタ
213: *              level  ･･･ ディレクトリの深さ
214: *    出　力 :  なし
```

〔リスト7-59〕プログラム d.c ④

```
215: *
216: ************************************************************************/
217: void dir (path, level)
218: char *path;
219: int level;
220: {
221:     int  file_count = 0;
222:     int  atr;
223:     char buff [PATH_LEN];
224:     DTA  dta;
225:     DTA  *dta_ptr;
226:     DIR  *dir_ptr, *ptr;
227:
228:     search_dir (path, level);
229:     dta_ptr = func_2f();                        /* ＤＴＡの退避 */
230:     func_1a (&dta);                             /* ＤＴＡの設定 */
231:     atr = R_flag ¦ B_flag ¦ S_flag ¦ A_flag;
232:     if (func_4e (path, atr)) {
233:         func_1a (dta_ptr);                      /* ＤＴＡの復帰 */
234:         if (! M_flag) {
235:             f_disp (file_count, level);
236:         }
237:         return;
238:     }
239:     do {
240:         if (dta.atr == atr ¦¦ dta.atr == ATR_ARC ¦¦ !dta.atr ) {
241:             file_count++;
242:         }
243:     } while (! func_4f ());                     /* ファイル数のカウント */
244:     if (! M_flag) {
245:         f_disp (file_count, level);
246:     }
247:     if (file_count) {
248:         af_count += file_count;
249:         dir_ptr = ptr = (DIR *) calloc (sizeof (DTA), file_count);
250:         if (ptr == NULL) {
251:             printf ("メモリが足りません.¥n");
252:             exit (2);
253:         }
254:         func_4e (path, atr);
255:         do {                                    /* ファイル情報の格納 */
256:             if (dta.atr == atr ¦¦ dta.atr == ATR_ARC ¦¦ !dta.atr) {
257:                 ptr -> atr = dta.atr;
258:                 strcpy (ptr -> name, dta.name);
259:                 ptr -> size = dta.size;
260:                 ptr -> time = dta.time;
261:                 ptr -> date = dta.date;
262:                 ptr++;
263:             }
264:         } while (! func_4f ());
265:         ptr -> atr = -1;
266:         if ((QF_flag ¦¦ QT_flag) && file_count >= 2) {
267:             qsort ((char *) dir_ptr, file_count, sizeof (DIR), dir_comp);
268:         }
269:         ptr = dir_ptr;
270:         do {
271:             disp_dir (ptr -> name, ptr -> atr, ptr -> size, ptr -> date,
272:                       ptr -> time, level);
273:             ptr++;
274:         } while (ptr -> atr != -1);
275:         free ((char *) dir_ptr);
276:     }
277:     func_1a (dta_ptr);                          /* ＤＴＡの復帰 */
278: }
279:
280: /************************************************************************
281: *
282: *    関数名 :  search_dir (path, level)
283: *    機  能 :  サブ・ディレクトリの読み込み
284: *    入  力 :  path  … パス名へのポインタ
285: *              level … ディレクトリの深さ
286: *    出  力 :  なし
287: *
288: ************************************************************************/
289: void search_dir (path, level)
290: char *path;
291: int level;
292: {
```

〔リスト7-59〕プログラムd.c ⑤

```
293:     char buff [PATH_LEN];
294:     char path_buff [PATH_LEN];
295:     char *str;
296:     DTA dta;
297:     DTA *dta_ptr;
298:     DIR *dir_ptr, *ptr;
299:     int file_count = 0;
300:
301:     dta_ptr = func_2f();                          /* ＤＴＡの退避 */
302:     func_1a (&dta);                               /* ＤＴＡの設定 */
303:     strcpy (path_buff, path);
304:     if (D_flag) {
305:         if (str = strrchr (path_buff, '¥¥')) {
306:             *++str = '¥0';
307:             strcat (path_buff, "*.*");
308:         }
309:     }
310:     if (! func_4e (path_buff, ATR_DIR)) {
311:         do {
312:             if (dta.atr == ATR_DIR &&
313:                 strcmpi (dta.name, ".") && strcmpi (dta.name, "..")) {
314:                 file_count++;
315:             }
316:         } while (! func_4f ());                   /* ディレクトリ数のカウント */
317:         if (! M_flag) {
318:             d_disp (file_count, level);
319:         }
320:         if (file_count) {
321:             ad_count += file_count;
322:             dir_ptr = ptr = (DIR *) calloc (sizeof (DTA), file_count);
323:             if (ptr == NULL) {
324:                 printf ("メモリが足りません.¥n");
325:                 exit (2);
326:             }
327:             func_4e (path_buff, ATR_DIR);
328:             do {                                  /* ディレクトリ情報の格納 */
329:                 if (dta.atr == ATR_DIR &&
330:                     strcmpi (dta.name, ".") && strcmpi (dta.name, "..")) {
331:                     ptr -> atr = dta.atr;
332:                     strcpy (ptr -> name, dta.name);
333:                     ptr -> size = dta.size;
334:                     ptr -> time = dta.time;
335:                     ptr -> date = dta.date;
336:                     ptr++;
337:                 }
338:             } while (! func_4f ());
339:             if ((QF_flag || QT_flag) && file_count >= 2) {
340:                 qsort ((char *) dir_ptr, file_count, sizeof (DIR), dir_comp);
341:             }
342:             ptr = dir_ptr;
343:             do {
344:                 disp_dir (ptr -> name, ptr -> atr, ptr -> size,
345:                           ptr -> date, ptr -> time, level);
346:                 if (D_flag) {
347:                     strcpy (buff, path);
348:                     if (str = strrchr (buff, '¥¥')) {
349:                         *++str = '¥0';
350:                     }
351:                     strcat (buff, ptr -> name);
352:                     strcat (buff, "¥¥");
353:                     strcpy (path_buff, path);
354:                     if (str = strrchr (path_buff, '¥¥')) {
355:                         strcat (buff, str + 1);
356:                     } else {
357:                         strcat (buff, "*.*");
358:                     }
359:                     dir (buff, level + 1);
360:                 }
361:                 ptr ++;
362:             } while (ptr -> atr);
363:             free ((char *) dir_ptr);
364:         }
365:     }
366:     func_1a (dta_ptr);                        /* ＤＴＡの復帰 */
367: }
368:
```

〔リスト7-59〕プログラム d.c ⑥

```
369: /***************************************************************
370: *
371: *    関数名:  vol_msg (drv)
372: *    機  能:  ボリューム名の表示
373: *    入  力:  drv ･･･ ドライブ名へのポインタ
374: *    出  力:  なし
375: *
376: ***************************************************************/
377: void vol_msg (drv)
378: char *drv;
379: {
380:     DTA  dta;
381:     DTA *dta_ptr;
382:     char buff [PATH_LEN];
383: 
384:     dta_ptr = func_2f ();
385:     func_1a (&dta);
386:     strcpy (buff, drv);
387:     strcat (buff, "¥¥*.*");
388:     printf ("ドライブ %c: のボリューム・ラベルは", *drv);
389:     if (func_4e (buff, ATR_VOL)) {
390:         printf ("ありません.¥n");
391:     } else {
392:         printf (" %s¥n", dta.name);
393:     }
394:     func_1a (dta_ptr);
395: }
396: 
397: /***************************************************************
398: *
399: *    関数名:  disp_dir (name, atr, size, date, time, level)
400: *    機  能:  ディレクトリの表示
401: *    入  力:  name  ･･･ パス名へのポインタ
402: *             atr   ･･･ ファイルの属性
403: *             size  ･･･ ファイルのサイズ
404: *             date  ･･･ 最終更新日付
405: *             time  ･･･ 最終更新時刻
406: *             level ･･･ ディレクトリ・レベル
407: *    出  力:  なし
408: *
409: ***************************************************************/
410: void disp_dir (name, atr, size, date, time, level)
411: char *name;
412: int atr;
413: long size;
414: unsigned int date, time;
415: int level;
416: {
417:     char f_name [NAME_SIZE + EXT_SIZE + 2];
418:     char *ptr;
419: 
420:     atr_disp (atr, level);                         /* 属性表示 */
421:     name_set (f_name, name);
422:     printf ("%s ", f_name);                        /* ファイル名表示 */
423:     if (atr == 0x10) {
424:         printf (" <DIR> ");
425:     } else {
426:         printf ("%7ld", size);                     /* サイズ表示 */
427:     }
428:     printf (" %d-%02d-%02d", (date >> 9) + 80,
429:                              (date >> 5) & 0x0F,
430:                              date & 0x01F);
431:     printf (" %02d:%02d¥n", time >> 11, (time >> 5) & 0x3F);
432: }
433: 
434: /***************************************************************
435: *
436: *    関数名:  atr_disp (atr, level)
437: *    機  能:  属性の表示
438: *    入  力:  atr   ･･･ ファイルの属性
439: *             level ･･･ ディレクトリ・レベル
440: *    出  力:  なし
441: *
442: ***************************************************************/
443: void atr_disp (atr, level)
444: int atr, level;
445: {
446:     while (--level) {
```

〔リスト7-59〕プログラムd.c ⑦

```
447:            printf ("  ");
448:        }
449:        printf ("¦-");
450:        if (atr & ATR_ARC) {
451:            printf ("a");
452:        } else {
453:            printf ("-");
454:        }
455:        if (atr & ATR_DIR) {
456:            printf ("d");
457:        } else {
458:            printf ("-");
459:        }
460:        if (atr & ATR_SYS) {
461:            printf ("s");
462:        } else {
463:            printf ("-");
464:        }
465:        if (atr & ATR_BRND) {
466:            printf ("b");
467:        } else {
468:            printf ("-");
469:        }
470:        if (atr & ATR_READ) {
471:            printf ("r");
472:        } else {
473:            printf ("-");
474:        }
475:        printf (" ");
476: }
477:
478: /***********************************************************************
479: *
480: *     関数名：    name_set (name_buff, pack_name)
481: *     機  能：    パックされたファイル名を12文字のファイル名に変換
482: *     入  力：    name_buff ･･･ 12文字のバッファへのポインタ
483: *                 pack_name ･･･ パックされたファイル名へのポインタ
484: *     出  力：    なし
485: *
486: ***********************************************************************/
487: void name_set (name_buff, pack_name)
488: char *name_buff, *pack_name;
489: {
490:        char *s_ptr, *d_ptr;
491:
492:        strcpy (name_buff, "            ");
493:        s_ptr = pack_name;
494:        d_ptr = name_buff;
495:        while (*s_ptr != '.' && *s_ptr != '¥0') {
496:            *d_ptr++ = *s_ptr++;                        /* ファイル名のコピー  */
497:        }
498:        d_ptr = name_buff + NAME_SIZE + 1;
499:        if ((s_ptr = strchr (pack_name, '.')) != NULL) {
500:            while (*++s_ptr != '¥0') {
501:                *d_ptr++ = *s_ptr;                   /* 拡張子のコピー  */
502:            }
503:        }
504: }
505:
506: /***********************************************************************
507: *
508: *     関数名：   f_disp (n, level)
509: *     機  能：   ファイル数の表示
510: *     入  力：   n       ･･･ ファイル数
511: *                level   ･･･ ディレクトリ・レベル
512: *     出  力：   なし
513: *
514: ***********************************************************************/
515: void f_disp (n, level)
516: int n, level;
517: {
518:        while (--level) {
519:            printf ("  ");
520:        }
521:        printf ("¦---%6d 個のファイルがあります.¥n", n);
522: }
523:
```

〔リスト7-59〕プログラム d.c ⑧

```
524: /*****************************************************************
525: *
526: *     関数名 :   d_disp (n, level)
527: *     機  能 :   ディレクトリ数の表示
528: *     入  力 :   n      ・・・ディレクトリ数
529: *                level ・・・ ディレクトリ・レベル
530: *     出  力 :   なし
531: *
532: *****************************************************************/
533: void d_disp (n, level)
534: int n, level;
535: {
536:        while (--level) {
537:            printf ("    ");
538:        }
539:        printf ("|---%6d 個のサブ・ディレクトリがあります.¥n", n);
540: }
541:
542: /*****************************************************************
543: *
544: *     関数名 :   af_disp ()
545: *     機  能 :   ファイル数とディレクトリ数の合計表示
546: *     入  力 :   なし
547: *     出  力 :   なし
548: *
549: *****************************************************************/
550: void af_disp ()
551: {
552:        printf ("|¥n");
553:        printf ("|---   全部で %d 個のサブ・ディレクトリがあります.¥n", ad_count);
554:        printf ("|---   全部で %d 個のファイルがあります.¥n¥n", af_count);
555: }
556:
557: /*****************************************************************
558: *
559: *     関数名 :   dir_comp (x, y)
560: *     機  能 :   ディレクトリのファイル名または日時を比較
561: *     入  力 :   x ・・・ 第一要素へのポインタ
562: *                y ・・・ 第二要素へのポインタ
563: *     出  力 :   x > y ・・・ 正
564: *                x = y ・・・ 0
565: *                x < y ・・・ 負
566: *
567: *****************************************************************/
568: int dir_comp (x, y)
569: DIR *x, *y;
570: {
571:         if (QT_flag) {
572:             if (x -> date > y -> date) {
573:                 return (1);
574:             }
575:             if (x -> date < y -> date) {
576:                 return (-1);
577:             }
578:             if (x -> time > y -> time) {
579:                 return (1);
580:             }
581:             if (x -> time < y -> time) {
582:                 return (-1);
583:             } else {
584:                 return (0);
585:             }
586:         } else {
587:             return (strcmp (x -> name, y -> name));
588:         }
589: }
```

● dsub.asm

リスト7-60はプログラム d.exe のアセンブリ・ソース部分です．同リストにおいて，サブルーチン(関数) func_19 は，ファンクション 19H を用いてカレント・ドライブ番号の読み出しを行い，そのドライブ番号を AX レジスタに返します．

サブルーチン func_1a は，ファンクション 1AH を用いて，引数 arg1 で指定された DTA の設定を行います．

サブルーチン func_2f は，ファンクション 2FH を用いて DTA の読み出しを行い，そのアドレスを AX レジスタに NEAR ポインタとして返します．

〔リスト7-60〕プログラム dsub.asm ①

```
 1: ;*********************************************************************
 2: ;
 3: ;    機    能 :  ディレクトリ表示サブルーチン
 4: ;  ファンクション :  19H(カレント・ドライブ番号の読み出し)
 5: ;                1AH(DTAアドレスの設定)
 6: ;                2FH(DTAアドレスの読み出し)
 7: ;                47H(カレント・ディレクトリの読み出し)
 8: ;                4EH(一致するファイルの検索)
 9: ;                4FH(つぎのファイルの検索)
10: ;    生    成 :  masm /ML dsub;
11: ;
12: ;*********************************************************************
13:          .MODEL  SMALL, C
14:          .CODE
15: ;*********************************************************************
16: ;
17: ;    ルーチン名 :  func_19
18: ;    機    能 :  カレント・ドライブ番号の取得
19: ;    func     :  19H(カレント・ドライブ番号の読み出し)
20: ;    入    力 :  なし
21: ;    出    力 :  AX ・・・ ドライブ番号(00H=A, 01H=B, ・・・)
22: ;
23: ;*********************************************************************
24: func_19    PROC
25:            mov      ah, 19h                ;ファンクション 19H
26:            int      21h
27:            xor      ah, ah
28:            ret
29: func_19    ENDP
30:
31: ;*********************************************************************
32: ;
33: ;    ルーチン名 :  func_1a
34: ;    機    能 :  DTAアドレス設定
35: ;    func     :  1aH(ディスク転送アドレスの設定)
36: ;    入    力 :  arg1  ・・・ DTAへのポインタ
37: ;    出    力 :  なし
38: ;
39: ;*********************************************************************
40: func_1a    PROC     arg1:PTR
41:            mov      dx, arg1
42:            mov      ah, 1Ah                            ;ファンクション 1AH
43:            int      21h
44:            ret
45: func_1a    ENDP
46:
47: ;*********************************************************************
48: ;
49: ;    ルーチン名 :  func_2f
50: ;    機    能 :  DTAアドレスの取得
51: ;    func     :  2FH(DTAアドレスの読み出し)
52: ;    入    力 :  なし
53: ;    出    力 :  AX ・・・ アドレスのオフセット部
54: ;
55: ;*********************************************************************
56: func_2f    PROC
57:            push     es
58:            mov      ah, 2Fh                ;ファンクション 2FH
59:            int      21h
60:            mov      ax, bx
61:            pop      es
62:            ret
63: func_2f    ENDP
64:
65: ;*********************************************************************
66: ;
67: ;    ルーチン名 :  func_47
68: ;    機    能 :  カレント・ディレクトリの読み出し
69: ;    func     :  47H(カレント・ディレクトリの読み出し)
70: ;    入    力 :  arg1  ・・・ ドライブ番号
71: ;                        (00H=カレント, 01H=A, 02H=B, ・・・)
72: ;                arg2 ・・・ バッファへのポインタ
73: ;    出    力 :  AX          00H: エラーなし
74: ;                            03H: ドライブ名が無効
75: ;
76: ;*********************************************************************
77: func_47    PROC     arg1:WORD, arg2:PTR
78:            push     si
```

〔リスト7-60〕プログラム dsub.asm ②

```
 79:               mov     dx, arg1
 80:               mov     si, arg2
 81:               mov     ah, 47h              ;ファンクション47H
 82:               int     21h
 83:               jc      err
 84:               xor     ax, ax
 85: err:
 86:               pop     si
 87:               ret
 88: func_47       ENDP
 89:
 90: ;**************************************************************
 91: ;
 92: ;    ルーチン名 :  func_4e
 93: ;    機   能 :  一致するファイルを検索し結果をDTAに返す
 94: ;    func   :  4EH(一致するファイルの検索)
 95: ;    入   力 :  arg1  … ファイル名へのポインタ
 96: ;              arg2  … 属性
 97: ;    出   力 :  AX    … エラーなし :  0
 98: ;                      エラーあり :  エラー・コード
 99: ;
100: ;**************************************************************
101: func_4e       PROC    arg1:PTR, arg2:WORD
102:               mov     dx, arg1
103:               mov     cx, arg2
104:               mov     ah, 4Eh              ;ファンクション4EH
105:               int     21h
106:               jc      error
107:               xor     ax, ax
108: error:
109:               ret
110: func_4e       ENDP
111:
112: ;**************************************************************
113: ;
114: ;    ルーチン名 :  func_4f
115: ;    機   能 :  つぎに一致するファイルを検索し結果をDATに返す
116: ;    func   :  4FH(つぎのファイル検索)
117: ;    入   力 :  なし
118: ;    出   力 :  AX    … エラーなし :  0
119: ;                      エラーあり :  エラー・コード
120: ;
121: ;**************************************************************
122: func_4f       PROC
123:               mov     ah, 4Fh              ;ファンクション4FH
124:               int     21h
125:               jc      error
126:               xor     ax, ax
127: error:
128:               ret
129: func_4f       ENDP
130:               END
```

サブルーチン func_47 は，ファンクション 47H を用いて，arg1 のドライブ番号で指定されたドライブのカレント・ディレクトリを読み出し，ポインタ arg2 で指定されたバッファ領域に格納します．ここで，もしファンクション 47H でエラーが発生すれば，そのエラー・コードを AX レジスタに返し，正常に終了した場合は AX レジスタに 0 を返します．

サブルーチン func_4e は，ファンクション 4EH を用いて，ポインタ arg1 で指定されたファイル名(パス名であり，ワイルド・カードを含む)と arg2 で指定された属性をもつファイルの検索を行い，その検索した結果を DTA で指定されているバッファ領域に格納します．ここで，もしファンクション 4EH でエラーが発生すれば，そのエラー・コードを AX レジスタに返し，正常に終了した場合は AX レジスタに 0 を返します．

サブルーチン func_4f は，ファンクション 4FH を用いて，次のファイルの検索を行います．ここでも，その検索した結果は DTA で指定されているバッファ領域に格納されます．また，もしファンクション 4FH でエラーが発生すれば，そのエラー・コードを AX レジスタに返し，正常に終了した場合は AX レジスタに 0 を返します．

◆ 生成方法

プログラム d.exe は，以下の手順で分割アセンブル/コンパイルして作成します．

```
masm /ML dsub;
cl −J −As d.c dsub −link /STACK:4096
```

cl コマンドの－link オプションは，リンカ LINK に対するオプションの指定を行うもので，ここでは生成される d.exe のスタック領域を 4096 バイトに指定します．

このオプションを指定しないと，ディレクトリ表示を行うための関数が再帰的に呼び出されるため，レベルの深いディレクトリを表示する際に，スタック・オーバフローのエラーが発生してしまいます．

◆ 実行サンプル

リスト7-61 はディレクトリ表示プログラム d.exe の実行例を示しています．

① まず，dir コマンドでドライブ H のルート・ディレクトリを確認する．

② サブ・ディレクトリやファイル名が表示される．

③ 同様に，プログラム d.exe でドライブ H のルート・ディレクトリを表示する．

④ このフィールドにはファイルの属性が表示される．

⑤ また，サブ・ディレクトリの個数も表示され，サブ・ディレクトリが先に表示されている．

⑥ そのあとに，ファイルの個数が表示されファイルがまとめて表示される．

⑦ 最後にサブ・ディレクトリの数とファイルの数が，それぞれ合計して表示される．

⑧ 再び，プログラム d.exe を用いてドライブ H のカレント・ディレクトリの表示を行う．この際に－h オプションを指定してシステム・ファイルや隠されたファイルの表示も行う．

⑨ dir コマンドでは表示されないファイルが表示される．ここで，ファイル属性が一般のファイルとは異なることに注目．

⑩ プログラム d.exe を用いてディレクトリの階層構造の表示を行う．ここでは－d オプションおよび－m オプションを指定している．また，ファイル名としてありえない“.”を指定していることに注意．

⑪ サブ・ディレクトリ名だけが表示される．

⑫ 次に，プログラム d.exe を用いてすべてのサブ・ディレクトリ内の拡張子 .c をもつファイルを検索して表示する．

⑬ 拡張子 .c のファイルだけが表示される．

⑭ サブ・ディレクトリの数とファイルの数の合計数が表示される．

⑮ 次に，dir コマンドでドライブ H のカレント・ディレクトリ内の拡張子 .asm のファイルを確認する．この場合は，ディレクトリに登録された順序に表示される．

⑯ プログラム d.exe に対して－qf オプションを指定して，ドライブ H のカレント・ディレクトリ内の拡張子 .asm のファイルを表示する．

⑰ －qf オプションによって名前でソートされて表示されている．

⑱ 次に，プログラム d.exe に対して－qt オプションを指定して，ドライブ H のカレント・ディレクトリ内にある拡張子 .asm のファイルを表示する．

⑲ －qt オプションによって更新日時でソートされて表示されている．

〔リスト7-61〕
プログラム d.exe の実行例 ①

```
R>dir h:¥   ⏎… ドライブHのルート・ディレクトリを確認①

 ドライブ H: のディスクのボリュームラベルは HD1
 ディレクトリは H:¥

AUTOEXEC BAK     366  88-11-08  14:34 ┐
COM1       <DIR>      88-11-05   9:43 │
COM3       <DIR>      88-11-05   9:43 │
                                      │
                                      │
                                      │
WK3        <DIR>      88-11-05   9:45 │
TERM       <DIR>      88-11-05  10:11 ├ サブ・ディレクトリとファイル名の表示②
AUTOEXEC BAT     361  88-12-05  13:55 │
GAIJ            9392  88-09-13   8:26 │
ATOK     DIC  563712  88-12-21  10:23 │
CONFIG   SYS     262  88-12-16  16:34 │
CONFIG   RAM     147  88-11-28  11:04 │
CONFIG   BAK     260  88-12-16  16:02 ┘
      25 個のファイルがあります.
   5169152 バイトが使用可能です.

R>d h:¥   ⏎… プログラム d.exe を用いてドライブ H のルート・ディレクトリを確認③

 *** ディレクトリ表示プログラム  Ver.1.1 ***

ドライブ H: のボリューム・ラベルは  HD1
```

〔リスト7-61〕
プログラム
d.exe の実行例 ②

```
H:¥ ···: ディレクトリ
|---      18 個のサブ・ディレクトリがあります.
|--d---  COM1          <DIR>   88-11-05 09:43  ┐
|--d---  COM3          <DIR>   88-11-05 09:43  │
   ↑          ↑                                │
(ファイル属性④) (ディレクトリ個数の表示)          ├ サブ・ディレクトリの表示⑤
                                               │
|--d---  WK3           <DIR>   88-11-05 09:45  │
|--d---  TERM          <DIR>   88-11-05 10:11  ┘
|---      7 個のファイルがあります.  ←(ファイルの数)
|-a----  AUTOEXEC BAK     366  88-11-08 14:34  ┐
|-a----  AUTOEXEC BAT     361  88-12-05 13:55  │
|-a----  GAIJ            9392  88-09-13 08:26  │
|-a----  ATOK     DIC  563712  88-12-21 10:23  ├ ファイルの表示⑥
|-a----  CONFIG   SYS     262  88-12-16 16:34  │
|-a----  CONFIG   RAM     147  88-11-28 11:04  │
|-a----  CONFIG   BAK     260  88-12-16 16:02  ┘

|---   全部で 18 個のサブ・ディレクトリがあります.  ←(合計の個数⑦)
|---   全部で 7 個のファイルがあります.

R>d h:¥ -h ⏎… システム·ファイルや隠されたファイルの表示⑧

*** ディレクトリ表示プログラム Ver.1.1 ***

ドライブ H: のボリューム・ラベルは HD1
H:¥ ···· ディレクトリ
|---      18 個のサブ・ディレクトリがあります
|--d---  COM1          <DIR>   88-11-05 09:43
|--d---  COM3          <DIR>   88-11-05 09:43
 ┌ ファイル属性に注目⑨
 │
|--d---  WK3           <DIR>   88-11-05 09:45
|--d---  TERM          <DIR>   88-11-05 10:11
|---      9 個のファイルがあります.
|-a-sbr  IO       SYS   49152  87-10-23 00:00  ┐ dir コマンドでは表示されない
|-a-sbr  MSDOS    SYS   28160  87-10-23 00:00  ┘
|-a----  AUTOEXEC BAK     366  88-11-08 14:34
|-a----  AUTOEXEC BAT     361  88-12-05 13:55
|-a----  GAIJ            9392  88-09-13 08:26
|-a----  ATOK     DIC  563712  88-12-21 10:23
|-a----  CONFIG   SYS     262  88-12-16 16:34
|-a----  CONFIG   RAM     147  88-11-28 11:04
|-a----  CONFIG   BAK     260  88-12-16 16:02

|---   全部で 18 個のサブ・ディレクトリがあります.
|---   全部で 9 個のファイルがあります.

R>d h:¥.. -d -m ⏎…ディレクトリの階層表示⑩

*** ディレクトリ表示プログラム Ver.1.1 ***

ドライブ H: のボリューム・ラベルは HD1
H:¥ ···· ディレクトリ
|--d---  COM1    ┐         <DIR>   88-11-05 09:43
|--d---  COM3    │ディレク  <DIR>   88-11-05 09:43
|--d---  SYS     │トリ名だ  <DIR>   88-11-05 09:44
|--d---  BAT     │け表示さ  <DIR>   88-11-05 09:44
|--d---  JXW     ┘れる⑪    <DIR>   88-11-05 09:44
    |--d---  UT            <DIR>   88-12-14 09:33
|--d---  HANA              <DIR>   88-11-05 09:44
|--d---  CAND3             <DIR>   88-12-21 12:15
|--d---  DB                <DIR>   88-11-05 09:44
    |--d---  RB            <DIR>   88-11-05 10:49
|--d---  TOOL              <DIR>   88-11-05 09:44
    |--d---  C             <DIR>   88-11-05 10:46
|--d---  MSC               <DIR>   88-11-05 09:44
    |--d---  EXE           <DIR>   88-11-05 10:50
    |--d---  INCLUDE       <DIR>   88-11-05 10:51
        |--d---  SYS       <DIR>   88-11-05 10:58
    |--d---  LIB           <DIR>   88-11-05 10:51
    |--d---  STARTUP       <DIR>   88-11-05 10:58
|--d---  FOR               <DIR>   88-11-05 09:44
|--d---  PAS               <DIR>   88-11-05 09:44
|--d---  WK                <DIR>   88-11-05 09:44
```

〔リスト7-61〕
プログラム
d.exe の実行例 ③

```
    |--d---LCSS          <DIR>  88-11-05 11:09
 |--d--- BIN          <DIR>  88-11-05 09:45
 |--d--- WK1          <DIR>  88-11-05 09:45
    |--d--- WK             <DIR>  88-11-08 15:16
    |--d--- SRC            <DIR>  88-11-05 12:44
 |--d--- WK2          <DIR>  88-11-05 09:45
 |--d--- WK3          <DIR>  88-11-05 09:45
 |--d--- TERM         <DIR>  88-11-05 10:11
    |--d--- ET             <DIR>  88-11-05 10:46
    |--d--- CT             <DIR>  88-11-05 10:46
 |
 |---  全部で 31 個のサブ・ディレクトリがあります.
 |---  全部で 0 個のファイルがあります.

R>d h:¥*.c -d -m  ⏎…すべてのディレクトリ内の.cファイルの表示⑫

 *** ディレクトリ表示プログラム Ver.1.1 ***

ドライブ H: のボリューム・ラベルは HD1
H:¥ ・・・・ ディレクトリ
 |--d--- COM1         <DIR>  88-11-05 09:43
 |--d--- COM3         <DIR>  88-11-05 09:43
 |--d--- SYS          <DIR>  88-11-05 09:44
 |--d--- BAT          <DIR>  88-11-05 09:44
 |--d--- JXW          <DIR>  88-11-05 09:44
    |--d--- UT         <DIR>  88-12-14 09:33
 |--d--- HANA         <DIR>  88-11-05 09:44
 |--d--- CAND3        <DIR>  88-12-21 12:15
 |--d--- DB           <DIR>  88-11-05 09:44
    |--d--- RB         <DIR>  88-11-05 10:49
 |--d--- TOOL         <DIR>  88-11-05 09:44
    |--d--- C          <DIR>  88-11-05 10:46
       |-a---- NUM       C      113 88-05-30 09:37
       |-a---- MSG       C      211 88-06-20 14:35   ← 拡張子.cのファイル⑬
 |--d--- MSC          <DIR>  88-11-05 09:44
    |--d--- EXE          <DIR>  88-11-05 10:50
    |--d--- INCLUDE      <DIR>  88-11-05 10:51
       |--d--- SYS           <DIR>  88-11-05 10:58
    |--d--- LIB          <DIR>  88-11-05 10:51
    |--d--- STARTUP      <DIR>  88-11-05 10:58
       |-a---- NULBODY   C       13 87-07-01 18:00
       |-a---- WILD      C     4863 87-07-01 18:00
 |--d--- FOR          <DIR>  88-11-05 09:44
 |--d--- PAS          <DIR>  88-11-05 09:44
 |--d--- WK           <DIR>  88-11-05 09:44
    |--d--- LCSS         <DIR>  88-11-05 11:09
    |-a---- LCT       C      3104 88-10-01 18:32
    |-a---- LCS       C      1959 88-09-07 09:10
    |-a---- LCG       C      1629 88-09-02 10:55
                 ・
                 ・
    |-a---- ERROR     C      4945 88-09-22 15:05
    |-a---- EXE       C      1092 88-10-25 13:25
    |-a---- LS1       C    139285 88-11-02 11:04
 |--d--- BIN          <DIR>  88-11-05 09:45
 |--d--- WK1          <DIR>  88-11-05 09:45
    |--d--- WK         <DIR>  88-11-08 15:16
       |-a---- CTRL      C     1366 88-12-12 09:16
       |-a---- FATAL     C     3620 88-12-12 10:07
       |-a---- DDUMP     C     6058 88-12-12 10:50
                 ・
                 ・
       |-a---- FCB       C     7438 88-12-14 08:09
       |-a---- UPPER     C      769 88-12-12 10:40
       |-a---- D         C    16078 88-12-21 14:48
    |--d--- SRC          <DIR>  88-11-05 12:44
       |-a---- TESTA     C       71 85-08-29 12:24
       |-a---- UP        C      661 85-10-07 14:24
                 ・
                 ・
       |-a---- LST       C      613 85-10-07 14:14
       |-a---- TAB       C      412 85-10-07 14:12
 |--d--- WK2          <DIR>  88-11-05 09:45
 |--d--- WK3          <DIR>  88-11-05 09:45
 |--d--- TERM         <DIR>  88-11-05 10:11
```

〔リスト7-61〕
プログラム
d.exe の実行例 ④

```
  |--d---ET           <DIR>  88-11-05 10:46
  |-a---- ETERM   C   10782 87-11-24 10:44
  |--d--- CT          <DIR>  88-11-05 10:46
|
|---  全部で 31 個のサブ・ディレクトリがあります.   ←合計数の表示⑭
|---  全部で 66 個のファイルがあります.

R>dir h:*.asm ⏎… ドライブHのカレント・ディレクトリ内の.asmファイルの確認⑮

 ドライブ H: のディスクのボリュームラベルは HD1
 ディレクトリは H:¥WK1¥WK

DIRSORT  ASM     4521  85-12-04  22:14
CTRLSUB  ASM     2242  88-12-12   9:24
STMPSUB  ASM     5310  88-12-09  10:27
                   .
                   .
GINTSUB  ASM      855  88-12-08  13:39
FCBSUB   ASM     6125  88-12-14   8:09
DSUB     ASM     3967  88-12-19  14:32
        47 個のファイルがあります.
   5169152 バイトが使用可能です.

R>d h:*.asm -qf ⏎… ドライブHのカレント・ディレクトリの.asmファイルの確認（名前でソート）⑯

 *** ディレクトリ表示プログラム Ver.1.1 ***

ドライブ H: のボリューム・ラベルは HD1
H:¥WK1¥WK¥ ･･･ ディレクトリ
|---      0 個のサブ・ディレクトリがあります.
|---     47 個のファイルがあります.
|-a---- CHANGE    ASM      977 88-12-02 09:52 ┐
|-a---- CHD       ASM      916 88-12-07 12:05 │
|-a---- CHDSUB    ASM     2191 88-12-16 11:09 │
                   .                          ├ファイル名でソートされる⑰
                   .                          │
|-a---- STMPSUB   ASM     5310 88-12-09 10:27 │
|-a---- TM        ASM     4539 88-12-12 11:12 │
|-a---- UPPERSUB  ASM      930 88-12-12 10:41 ┘
|
|---  全部で 0 個のサブ・ディレクトリがあります.
|---  全部で 47 個のファイルがあります.

R>d h:*.asm -qt ⏎… ドライブHのカレント・ディレクトリ内の.asmファイルの確認（日時でソート）⑱

 *** ディレクトリ表示プログラム Ver.1.1 ***

ドライブ H: のボリューム・ラベルは HD1
H:¥WK1¥WK¥ ･･･ ディレクトリ
|---      0 個のサブ・ディレクトリがあります.
|---     47 個のファイルがあります.
|-a---- CODESUB   ASM      748 88-12-07 08:44  ←日時でソートされる⑲
|-a---- PRAUXSUB  ASM     1034 88-12-07 09:37
|-a---- DIV0      ASM     1534 88-12-07 09:55
|-a---- FCBSUB    ASM     6125 88-12-14 08:09
|-a---- CHDSUB    ASM     2191 88-12-16 11:09
|-a---- DSUB      ASM     3967 88-12-19 14:32
|
|---  全部で 0 個のサブ・ディレクトリがあります.
|---  全部で 47 個のファイルがあります.

R>
```

*　　　　　　　　*

この章では，前の第6章で述べたMS-DOSのファンクション・リクエストを利用し，プログラム実例を示してきました．

これらのプログラムでは，できるだけMS-DOS標準のファンクション・リクエストを利用した「行儀のよいソフト」を心がけました．MS-DOSのファンクション・リクエストには，種類(機能)が豊富に揃っているので，ユーザのアイデアしだいでは，まだまだ利用価値の高いコマンドを作成することができます．

ここで示したプログラム群は，あくまでもファンクション・リクエストの一応用例であり，本書の読者諸兄もアイデアをしぼって，もっともっと便利なコマンドの開発に挑戦してほしいものです．

第III部

デバイス・ドライバと拡張メモリ

第II部では，MS-DOS から提供されている数々の機能を利用するための知識として，ファンクション・リクエストの使用方法と，その応用プログラム例を示してきました．

第III部では，逆にMS-DOS にユーザの用意したデバイス(I/O)を接続する際に必要となる知識として，デバイス・ドライバの構造について解説します．

MS-DOS では，一定のルールに従ったプログラム(ドライバ)を作成することによって，ユーザ・デバイスを簡単にOSの管理下におくことが可能になっています．第8章では，このデバイス・ドライバ作成のルールについて詳しく解説しています．

また第9章では，MS-DOS ver.3.30 でサポートが開始された拡張メモリについて，その必要性と構造，および利用方法について解説していきます．拡張メモリは，誕生して間もないMS-DOSの新しい強化機能です．拡張メモリには，MS-DOS の将来を占うだけの非常に興味深いものを感じます．将来のMS-DOSで標準になるであろう拡張メモリを，この章を参考にしてアクセスしてみようではありませんか．

Appendix には，二つのデバイス・ドライバの例を示してあります．ソフトウェアの解説というものは，文章と図表だけですべてを伝えるのは困難です．そこで，ここでも実用的に価値のあるプログラムとしてデバイス・ドライバの実例を掲げて解説し，少しでも理解を助けるように心がけました．

第8章 MS-DOSのデバイス・ドライバ

デバイス・ヘッダとコマンド・パケットとI/Oリクエスト

本章では，MS-DOSのデバイス・ドライバの作成方法について解説します．デバイス・ドライバとは，MS-DOSが標準でサポートしているデバイス(I/O装置)以外のデバイスを制御するためのプログラムのことです．

MS-DOSでは，ユーザの要求するデバイスを追加する際に，その制御プログラムをio.sysなどのBIOSに組み込んだり改造したりすることなしに，システムの拡張を行うことが可能になっています．すなわち，そのデバイスに対するドライバ(プログラム)をユーザが作成し，その名前をconfig.sysファイルにエディタを使って登録するだけで，次のブート時からこれらのユーザ・デバイスが使用可能となります．このようにして登録された新デバイスは，たとえばcopyコマンドのドライブ名とするなど，通常のコマンド・レベルでの操作が可能になります．

ここでは，デバイス・ドライバの実現を可能にするため，まずデバイス・ドライバの一般的な構造について解説していきます．デバイス・ドライバの実例としては，Appendix Bでグラフィックス・コンソール・ドライバを，Appendix CではRAMディスク・ドライバをとりあげ，全ソース・リストとともに解説しています．

8-1 構造と呼び出しの手順

ここでは，MS-DOSのデバイス・ドライバの構造と，その呼び出し手順について解説します．

デバイス・ドライバの呼び出し

第3章でも述べたように，MS-DOSはシステムのブート時にconfig.sysファイルを読みにいきます．そして，もしここにユーザのデバイス・ドライバが登録されていると，そのデバイス・ドライバのファイルを指定されたドライブ(パス)から読み込んで，メモリ中のMS-DOS本体(msdos.sys)の次のエリアにロードします(図8-1)．ここで，io.sysも実はメーカの供給するデバイス・ドライバの集まりになっています．

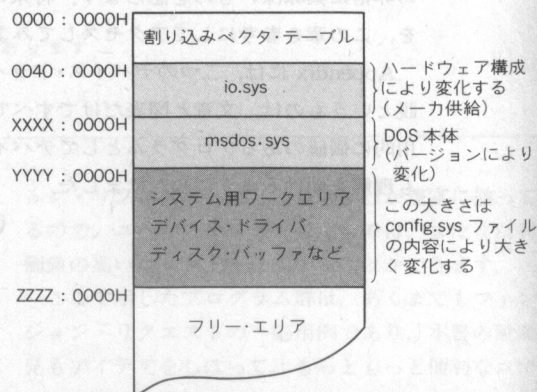
MS-DOSでは，デバイスの種類を“キャラクタ・デバイス”と“ブロック・デバイス”の二つに分類していますが，デバイス・ドライバの呼び出しシーケンス上では，これらの区別は特にありません．

キャラクタ・デバイスとは，コンソールやプリンタ，あるいは通信ポート(RS-232C)などのように，一度に1バイトのデータ転送を行うデバイスをいいます．

ブロック・デバイスとは，ディスクのようにデータ転送をブロック単位で行い，ランダム・アクセス可能なデバイスを指し，RAMディスクなどは後者に該当することになります．

デバイス・ドライバの構造は，図8-2に示すようにデバイス・ヘッダと各処理ルーチン(ストラテジ・ルーチンと割り込みルーチン)から構成されます．MS-DOSは，このデバイス・ヘッダという，デバイス・ドライバの先頭に置かれたテーブルを介してデバイス・ドライバをコールします．

MS-DOSは，たとえばRAMディスクがCドライブとして登録されているときに，

〔図8-1〕システム起動時のメモリ・マップ

copy　b:test.txt　c:

などのコマンド操作によって，登録されたデバイスへの入出力要求が起動されると，まずコマンド・コードや各種のパラメータを設定した“コマンド・パケット”(図8-5)を作成します．そして，このコマンド・パケットの先頭アドレスをES：BXレジスタにセットし，デバイス・ヘッダのストラテジ・ルーチンへのポインタを参照してストラテジ・ルーチンをFARコールします．デバイス・ドライバの呼び出し手順を図示すると，図8-3のようになります．

ストラテジ・ルーチンでは，このES：BXレジスタに入っているコマンド・パケットの先頭アドレスを，デバイス・ドライバのワーク・エリアに格納し，一度MS-DOSへFARリターンします．

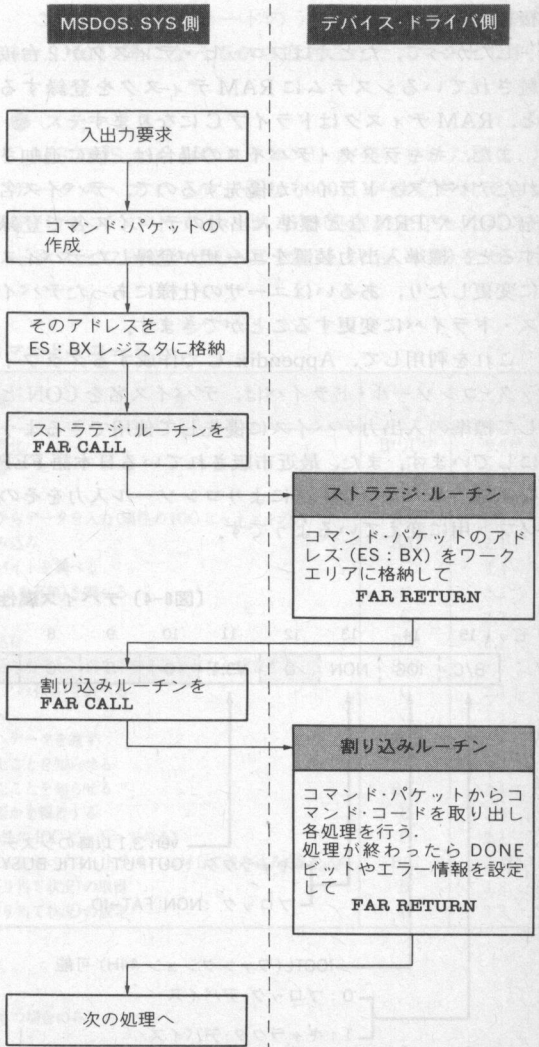
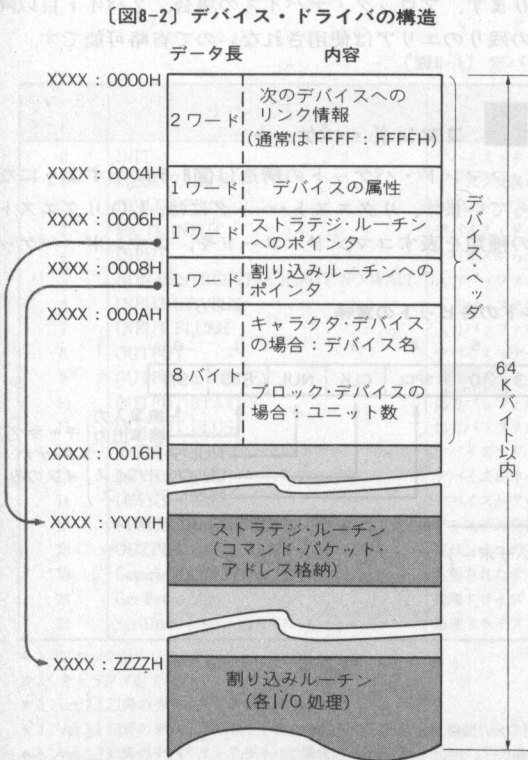
次に，MS-DOSはデバイス・ヘッダのポインタを参照して，今度はデバイス・ヘッダのプログラム本体である割り込みエントリ・ルーチンをFARコールします．

ここで，デバイス・ドライバは，先ほどのストラテジ・ルーチンで格納しておいたコマンド・パケットへのポインタを取り出し，これを参照してコマンド・パケット内のコマンド・コードをみて，各々のコマンドの処理を行います．そして，これらの処理が終了したら，コマンド・パケット内のステータス・ワードへDONEビット，あるいはエラー情報をセットしてFARリターンします．

ここでMS-DOSが，コマンド・パケットと二つのエントリ・ポインタを介してI/Oリクエストを発行している意義については，一説によると，将来MS-DOSがマルチタスク版になった場合を考慮した設計になっているというもので，現在のところはあまり意味はないようです．

これらの二つのルーチンでは，レジスタはすべて保存しなければなりません．また，MS-DOSが用意しているスタック領域は20個(ワード)程度なので，これ以上のスタックを必要とする場合は，デバイス・ドライバ側でスタック領域を確保しなければなりません．

〔図8-2〕デバイス・ドライバの構造

〔図8-3〕デバイス・ドライバの呼び出し

デバイス・ヘッダ

デバイス・ヘッダは，オフセット・アドレスの0000Hから始めなければなりません(通常のプログラムのようにORG 0100Hを付けてはいけない).

● リンク情報

デバイス・ドライバは，リンク情報によって複数のドライバがチェイン接続されます．MS-DOSは，このデバイス・ドライバのリンク情報をたどることにより，目的のドライバをサーチします．

通常，このフィールドにはプログラム時にFFFFH(−1)を設定しておきます．MS-DOSは，デバイス・ドライバを登録し，後述のINITコマンドが正常に終了した時点で，このフィールドに直前のデバイス・ドライバのアドレスを設定します．このため，ユーザが追加したデバイス・ドライバは，論理的にio.sysの後に接続されます．

したがって，たとえばフロッピ・ディスクが2台接続されているシステムにRAMディスクを登録すると，RAMディスクはドライブCになります．

また，キャラクタ・デバイスの場合は，後に追加されたデバイス・ドライバが優先するので，デバイス名をCONやPRNなど標準入出力のデバイス名で登録すると，標準入出力装置をユーザが登録したデバイスに変更したり，あるいはユーザの仕様にあったデバイス・ドライバに変更することができます．

これを利用して，Appendix Cで作成するグラフィック・コンソール・ドライバは，デバイス名をCONとして標準の入出力デバイスに優先して使用できるようにしています．また，最近市販されている日本語FEPなどは，このような手法によりコンソール入力をそのソフト用に変えているようです．

● デバイスの属性

デバイス属性フィールドは，デバイス・ドライバのタイプをMS-DOSに知らせるためにあります．各ビットの意味は図8-4のようになっています．

● ストラテジおよび割り込みルーチンへのポインタ

次の1ワードはストラテジ・ルーチンの，その次の1ワードは割り込みエントリ・ルーチンのオフセット・アドレスが入ります．これらの各ルーチンについてはあとで詳しく解説します．

ここで注意することは，この二つのポインタにはオフセット・アドレスしか入らないので，これらのルーチンはデバイス・ヘッダと同じセグメント内に配置されていなければならないということです．

したがって，コード部分が64Kバイトを越えるようなデバイス・ドライバの場合は，各々のルーチンからFARコールするか，オーバレイでディスクから読み込むなどの方法をとることになります．

● デバイス名またはユニット数

キャラクタ・デバイスの場合は，次の8バイトにデバイス名をASCII文字列でセットします．

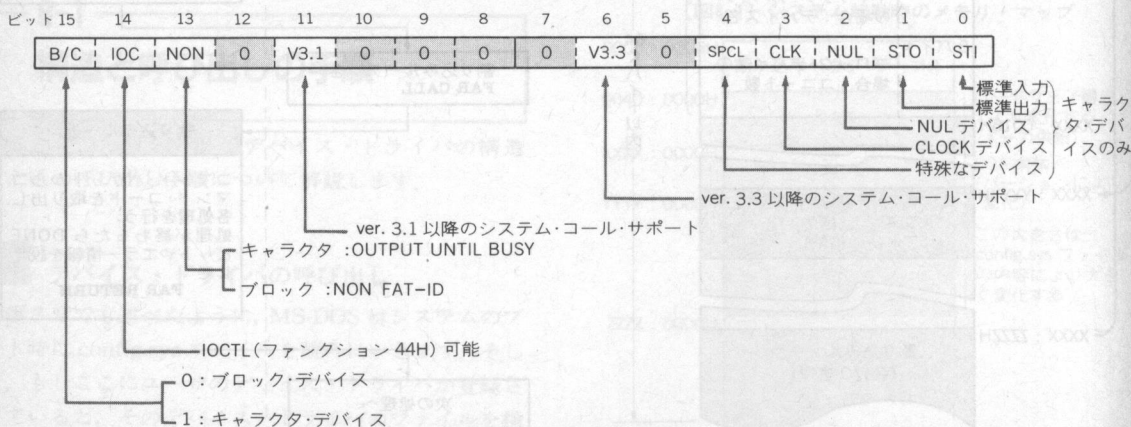
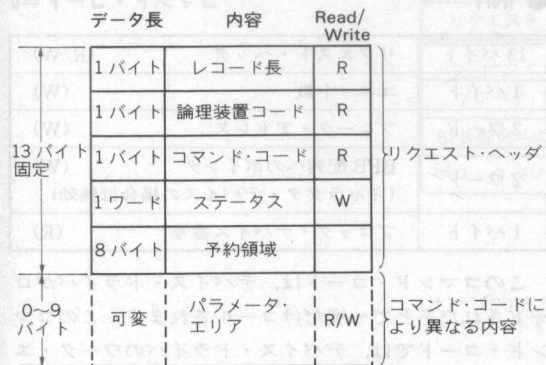
ブロック・デバイスの場合は，サポートするデバイスのユニット数(ドライブ数)が入ります．たとえば，4台のドライブをサポートするのであれば04Hになります．ブロック・デバイスの場合，2バイト目以降の残りのエリアは使用されないので省略可能です．

コマンド・パケット

コマンド・パケットの構造は図8-5に示すようになっています．リクエスト・ヘッダには，I/Oリクエストの種類を表すコマンド・コードや，コマンド・パケッ

〔図8-4〕デバイス属性フィールドの各ビットの意味

〔図8-5〕コマンド・パケット

	データ長	内容	Read/Write	
13バイト固定	1バイト	レコード長	R	リクエスト・ヘッダ
	1バイト	論理装置コード	R	
	1バイト	コマンド・コード	R	
	1ワード	ステータス	W	
	8バイト	予約領域		
0〜9バイト	可変	パラメータ・エリア	R/W	コマンド・コードにより異なる内容

トの長さなどが入っており，13バイトの固定長になっています．パラメータ・エリアは，各コマンドによりパラメータの数が違うので可変長になり，その内容も異なります．

● レコード長

レコード長のフィールドには，リクエスト・ヘッダ(13バイト)とパラメータ・エリア(可変長)を加えたコマンド・パケット全体の長さをバイト数で表現した数値が入ります．

● 論理装置コード

ブロック・デバイスの場合には，一つのドライバで複数のデバイス(ドライブ)をサポートすることが可能になっています．論理装置コードとは，I/Oリクエストがどのデバイス(ドライブ)に対して行われたのかを区別するためのコードです．

たとえば，ドライバが4台のディスク・ドライブをサポートしている場合に，0〜3のいずれかの値がMS-DOSによってセットされるので，デバイス・ドライバ側では，どのドライブに対してアクセスすべきかをこのコードにより決定できることになります．

● デバイス・コマンド・コード

I/Oリクエスト・コマンドの一覧を表8-1に示しておきます．これら20種類のI/Oリクエストのいずれかが，MS-DOSから0〜24のコマンド・コードにより，デバイス・ドライバに対して指示されます．

このコマンド・コードのうち，13〜24はMS-DOS ver.3.10以降になってから追加されたコマンドです．

● ステータス

MS-DOSからI/Oリクエストが発行された際に，ステータス・フィールドには0000Hが設定されてきます．デバイス・ドライバは，I/O処理が完了したらステータス・ワードのDONEビット(ビット8)をセット

〔表8-1〕デバイス・コマンド・コード

コマンド・コード	ファンクション名	機能	ブロック/キャラクタ	バージョン
0	INIT	デバイス・ドライバの初期化	B*1/C*2	2.1*3
1	MEDIA CHECK	ディスク交換の有無の調査	B	2.1
2	BUILD BPB	BPBテーブルの作成	B	2.1
3	IOCTL INPUT	デバイス・ドライバ自身からデータを入力(属性のIOCビット＝1のみ)	B/C	2.1
4	INPUT	デバイスからのデータ読み込み	B/C	2.1
5	NON-DESTRUCTIVE INPUT NO WAIT	入力バッファの先頭の1バイトを調べる	C	2.1
6	INPUT STATUS	入力バッファの状態(データの有無)を調べる	C	2.1
7	INPUT FLUSH	入力バッファを空にする	C	2.1
8	OUTPUT	デバイスへデータを書き込む	B/C	2.1
9	OUTPUT & VERIFY	デバイスへデータを書き込んだあと，正しく書き込まれたか検査する	B/C	2.1
10	OUTPUT STATUS	出力バッファの状態(データの有無)を調べる	C	2.1
11	OUTPUT FLUSH	出力バッファを空にする	C	2.1
12	IOCTL OUTPUT	デバイス・ドライバ自身へデータを渡す	B/C	2.1
13	DEVICE OPEN	デバイスがオープンされたことを知らせる	B/C	3.1*4
14	DEVICE CLOSE	デバイスがクローズされたことを知らせる	B/C	3.1
15	REMOVABLE MEDIA	ディスクの交換が可能か否かを報告する	B	3.1
16	OUTPUT UNTIL BUSY	プリンタへの連続出力(属性のIOCビット＝1のみ)	C	2.1
19	Generic IOCTL	拡張されたデータ(IOCTL)をデバイス・ドライバに対して読み込み/書き込み	B/C	3.3*5
23	Get Drive Map	論理ドライブ・マップ(割り当て状況)の取得	B	3.3
24	Set Drive Map	論理ドライブ・マップ(割り当て状況)の設定	B	3.3

＊1：ブロック型デバイス
＊2：キャラクタ型デバイス
＊3：ver.2.1以降のデバイス・ドライバ
＊4：ver.3.1以降のデバイス・ドライバでOPEN/CLOSE/RM機能(ver.3.1ビット)をもつ場合のみ
＊5：ver.3.3以降のデバイス・ドライバで属性のver.3.3ビットが"1"の場合のみ

〔図8-6〕ステータス・フィールドの各ビットの意味

ビット	15	14–10	9	8	7–0
	ERR	予備	BUSY	DONE	エラー・コード（ビット15=1で有効）

〔表8-2〕エラー・コード

エラー・コード	内　　容
00H	ライト・プロテクト違反
01H	無効なユニット
02H	ドライブの準備ができていない
03H	無効なコマンド・コード
04H	CRC エラー
05H	不正なコマンド・パケットの長さ
06H	シーク・エラー
07H	無効なメディア
08H	セクタが存在しない
09H	プリンタの用紙切れ
0AH	書き込みエラー
0BH	読み出しエラー
0CH	一般的なエラー
0DH	（予備）
0EH	（予備）
0FH	不正なディスク交換

して MS-DOS へ FAR リターンします．ステータス・ワードのビット・マップは**図8-6**のように定義されています．

また，I/O 処理中にエラーが発生した場合は，このフィールドには DONE ビットと ERR ビット，およびエラー情報をセットしなければなりません．エラー情報は，**表8-2**に示すように 16 種類が定義されています．

BUSY ビットは，コマンド・コード 6 または 10 の処理でセットしますが，その他のコマンド・コードでは 0 を返します．

8-2 I/O リクエスト・コマンド

デバイス・ドライバでは，**表8-1**に示したように全部で 20 個のコマンド・リクエストを処理しなければなりません．以下，各コマンド・コードについて解説していきます．

コマンド・パケットのうち，リクエスト・ヘッダは，すでに述べたように 13 バイトの固定長で，各コマンド・コードに共通なので，ここではコマンド・コードにより可変になるパラメータ・エリアについて述べることにします．またダブル・ワードによるポインタは，最初にオフセット，次にセグメントの順で入ります．

各項の R/W は，デバイス・ドライバが読むパラメータ(R)と，MS-DOS へ渡すためにデバイス・ドライバ側で設定するパラメータ(W)を示します．

● INIT　　　　コマンド・コード＝0

13 バイト	リクエスト・ヘッダ	(R/W)
1 バイト	ユニット数	(W)
2 ワード	ブレーク・アドレス	(W)
2 ワード	BPB 配列へのポインタ（キャラクタ・デバイスの場合は無効）	(W)
1 バイト	ブロック・デバイス番号	(R)

このコマンド・コードは，デバイス・ドライバがロードされたあとで一度だけコールされます．このコマンド・コードでは，デバイス・ドライバのワーク・エリアやデバイス自身の初期化を行ったあと，次の三つのパラメータをセットしてから MS-DOS に戻ります．

◆ ユニット数

ユニット数は，このデバイス・ドライバの扱うデバイスがブロック・デバイスの場合，サポートしているドライブ数をセットします．キャラクタ・デバイスの場合は“1”になります．

◆ ブレーク・アドレス

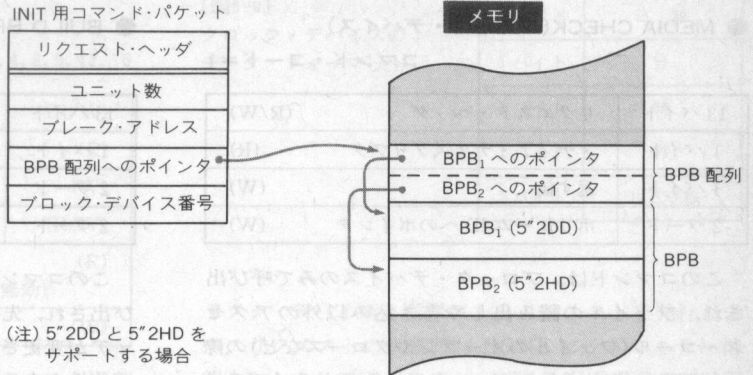
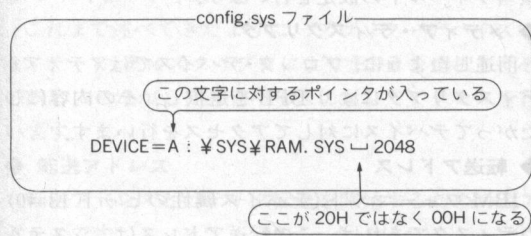
ブレーク・アドレスには，デバイス・ドライバの終了アドレスの次の解放できるアドレスをセットします．

このアドレス以降は，MS-DOS によって次のデバイス・ドライバやプログラム，あるいはディスク・バッファなどの領域として使用されます．したがって，一度限りで不用になる INIT コマンド処理ルーチンをデバイス・ドライバの最後部に置き，その開始アドレスをブレーク・アドレスにセットして返すことによってメモリの節約も可能になります．

◆ BPB 配列に対するポインタ

キャラクタ・デバイスの場合は必要ありません．このフィールドには，MS-DOS によって INIT コマンド・コードをコールする際に，**図8-7**のように config.sys ファイル中の DEVICE コマンドの“＝”の次の文字に対するポインタが設定されてきます．したがって，これを利用してデバイス・ドライバでは，種々のオプションに対する処理を行うことが可能です．

オプション処理を可能にしておくと，たとえば RAM ディスクなどで容量を変更したい場合に，デバイス・ドライバのソース・プログラムを再アセンブルすることなく，エディタによって，単に config.sys ファイルの内容を書き換えるだけで，その容量やモードを変更できることになります．

ここで注意したいことは，config.sys ファイル中の DEVICE コマンドで書かれたパラメータ文字列の空白コード(20H)は，MS-DOS により INIT コマンドに渡されるとき，同図のように NULL キャラクタ(00H)に変換されてきます．したがって，このパラメータ文

〔図8-7〕DEVICE コマンドのパラメータ文字列

〔表8-3〕BPB テーブル

1ワード	1セクタ当たりのバイト数
1バイト	1アロケーション・ユニット当たりのセクタ数
1ワード	予備のセクタ数
1バイト	FAT の数
1ワード	ルート・ディレクトリの数
1ワード	論理イメージのセクタ数
1バイト	メディア・ディスクリプタ
1ワード	1FAT 当たりのセクタ数

字列の解析を行う場合に，空白コードで判断していると不都合が生じてしまいます．

次に，このフィールドに返す値ですが，ブロック・デバイスの場合 BPB(BIOS Parameter Block)配列へのポインタを返します(図8-8)．ここで注意すべきことは，このフィールドは BPB 配列へのポインタであり，BPB の直接のアドレスを返してはいけないということです．

たとえば，ディスク・ドライバが 5″2HD と 5″2DD の異なるメディアをサポートする場合，ディスクのフォーマットが異なれば2種類の BPB を用意することになります．この場合に，各 BPB へのワード・オフセット(ポインタ)のテーブルを用意します．このポインタ・テーブルのことを BPB 配列と呼びます．

そして，コマンド・パケットの BPB 配列へのポインタ・フィールドには，この BPB 配列のダブル・ワードのアドレスを設定して返します．

BPB とは，FAT やディレクトリの大きさ，セクタの長さなどディスクに関する諸情報(第4章参照)をセットしたテーブルのことであり，表8-3 に示したような内容になっています．

ブロック・デバイスでは，一つのドライブで何種類ものメディア(たとえば 5″2DD と 5″2HD)をサポートする場合に，その種類の数だけ BPB が存在することになります．BPB テーブル内のメディア・ディスクリプタとは，これらの BPB(メディア)のうちどのタイプのメディアに対してアクセスを要求しているのかを判定するために使用されるもので，00～FFH の1バイトの値をとります．

このメディア・ディスクリプタを BPB の中にセットする場合，デバイス・ドライバは，任意の値をとることができますが，異なった BPB には異なったメディア・ディスクリプタを設定しなければなりません．

◆ブロック・デバイス番号

ブロック・デバイス番号は，ブロック型デバイスのみで利用でき，そのデバイスの論理的な番号(00H＝A，01H＝B，…)が渡されます．

◆FAT-ID

通常，RAM ディスクなどでは，INIT コマンドで FAT およびルート・ディレクトリ領域のメモリ・クリアを行います(これが RAM ディスクのフォーマッティングに当たる)．

このディスクのフォーマットを行ったのち，IBM フォーマット・ビット＝0 にセットしてあっても，FAT-ID をセットしておかないと，chkdsk など一部のコマンドへの対応ができないので注意が必要です．

● MEDIA CHECK(ブロック・デバイス)

コマンド・コード=1

13バイト	リクエスト・ヘッダ	(R/W)
1バイト	メディア・ディスクリプタ	(R)
1バイト	返す値	(W)
2ワード	ボリュームIDへのポインタ	(W)

このコマンドは，ブロック・デバイスのみで呼び出され，ファイルの読み出しや書き込み以外のアクセス・コール(ファイルのオープンやクローズなど)の際に使用されます．このコマンドは，そのドライブのメディアが変更されているかどうかを調べるために呼ばれます．

◆ メディア・ディスクリプタ

MS-DOSは，BPBで指定してあるメディア・ディスクリプタのなかから，調べようとするメディアのタイプに合ったメディア・ディスクリプタを探し，このフィールドに設定してからこのコマンドをリクエストします．

◆ 返す値

このコマンドでは，単にメディアが交換されたかどうかのチェックを以下の値で返します．

−1：メディアが交換された

0：メディアが交換されたかどうかわからない

1：メディアは交換されていない

通常のフロッピ・ディスクの場合は，ドア・オープンによってこの判定を行います．RAMディスクやハード・ディスクでは，メディアの交換はありえないので常に"1"(メディアは交換されていない)を返すことになります．

◆ ボリュームIDへのポインタ

デバイス属性のビット11が"1"(ver.3.10以降)で，かつこのコマンドで返す値フィールドに"−1"(メディア変更)を返した場合は，このフィールドに対してボリュームIDへのダブル・ワードのポインタを設定します．

このとき，もしドライバがボリュームIDをサポートしていない場合でも，

DB 'NO NAME', 0

へのポインタを設定する必要があります．

● BUILD BPB(ブロック・デバイス)

コマンド・コード=2

13バイト	リクエスト・ヘッダ	(R/W)
1バイト	メディア・ディスクリプタ	(R)
2ワード	転送アドレス	(R)
2ワード	BPBに対するポインタ	(W)

このコマンドは，ブロック・デバイスの場合のみ呼び出され，先行するMEDIA CHECKコマンドでメディアが変更されていた場合や，メディアが変更された可能性のある場合に任意の時点で使用されます．

このコマンドが発行されたら，デバイス・ドライバはBPBテーブルを作成してコマンド・パケット内の該当フィールドの設定を行います．

◆ メディア・ディスクリプタ

前述したように，ブロック・デバイスではメディア・ディスクリプタによりBPBを選択し，その内容にしたがってデバイスに対してアクセスを行います．

◆ 転送アドレス

IBMフォーマット(デバイス属性のビット13=0)のディスクであれば，この転送アドレスは，システム内の1セクタのバッファ・アドレスを指し，そのバッファにはFATの先頭セクタが入っています．

FATの第1バイトには第4章で解説したように，メディアのタイプによって決まっているFAT-IDバイトがセットされています．デバイス・ドライバは，このFAT-IDバイトを参照することによって，どのBPBを使用すべきかがすぐに判断できるようになっています．この場合に，デバイス・ドライバはシステム内のバッファの内容を書き換えてはいけません．

non-IBMフォーマット・ディスク(デバイス属性のビット13=1)の場合は，このバッファを必要に応じてデバイス・ドライバのワーク・エリアとして自由に使用できます．ただし，一時的なワーク・エリアとしてしか使用できません．また，この場合にFAT-IDは使用されないので，BPBの選択はデバイス・ドライバのみで行わなければなりません．

◆ BPBに対するポインタ

デバイス・ドライバは，メディアのタイプを調べてそれに合うBPBを作成または選択し，そのBPBのアドレス(ダブル・ワード)をBPBに対するポインタのフィールドにセットします．

ここで，このフィールドのポインタは，INITコマンドとは異なりBPB配列へのポインタでなく，BPBへの直接のポインタであることに注意が必要です．

● READ, WRITE & VERIFY

コマンド・コード＝3, 4, 8, 9, 12, 16

13バイト	リクエスト・ヘッダ	(R/W)
1バイト	メディア・ディスクリプタ	(R)
2ワード	転送アドレス	(W)
1ワード	バイト/セクタ・カウント	(R/W)
1ワード	開始セクタ番号 (キャラクタ・デバイスでは無効)	(R)
2ワード	ボリュームIDへのポインタ	(W)

これらのコマンドでは，ブロック・デバイスの場合にセクタ単位で読み書きを行い，キャラクタ・デバイスの場合にはバイト単位で読み書きが行われます．

◆ メディア・ディスクリプタ

これまで述べてきたように，ブロック・デバイスではメディア・ディスクリプタによりBPBを選択し，その内容にしたがって，デバイスに対してアクセスを行います．

◆ 転送アドレス

転送アドレスとはデータ転送を開始するための実アドレスです．

◆ バイト/セクタ・カウント

バイト/セクタ・カウントは，ブロック・デバイスの場合に転送するセクタ数を表します．キャラクタ・デバイスの場合には転送するバイト数を表し，通常1バイトになります．また，キャラクタ・デバイスのIOCTLコマンドでは，転送するIOCTL文字列の長さを表します．

これらの転送処理が終了したら，ドライバは実際に転送されたバイト数またはセクタ数を，このフィールドに設定して返します．

エラーなく終了したらそのままで返します．なぜなら，

要求されたバイト/セクタ＝転送したバイト/セクタ

となるからです．

エラーが生じたらエラー情報だけでなく，このフィールドに実際に転送された数を正しくセットして返してやらなければなりません．

◆ 開始セクタ番号

開始セクタ番号とは，データの転送を開始する論理セクタ番号です．キャラクタ・デバイスの場合は，このフィールドには意味がありません．

◆ クロック・デバイス

キャラクタ・デバイスの特殊なデバイスとしてクロック・デバイスがあります．システムがリアルタイム・クロックをサポートしていて日付や時刻を得たい場合に，MS-DOSはクロック・デバイスに対して他のキャラクタ・デバイスと同様にI/Oリクエストを発行します．リクエストの大部分はDONEビットやエラー情報をセットして返しますが，READ/WRITEでは，図8-9のようにフォーマットされた6バイトのデータを転送します．

〔図8-9〕
クロック・デバイスの
データ・フォーマット

2バイト	日　数
1バイト	分
1バイト	時
1バイト	秒
1バイト	1/100秒

● NON-DETRUCTIVE READ & NO-WAIT

(キャラクタ・デバイス)　コマンド・コード＝5

13バイト	リクエスト・ヘッダ	(R/W)
1バイト	デバイスからのデータ	(W)

このコマンドは，キャラクタ・デバイスによって，ステータス・ワードのBUSYビット＝0(バッファ内に文字が存在する)が返された場合に，これを先読みするためにコールされます．

◆ デバイスからのデータ

先読みしたデータ(文字)は，コマンド・パケット内のデバイスからのデータ・フィールドにセットします．このデータ(文字)は入力バッファから削除できません(NON-DESTRUCTIVE)．

また，先読みするデータがない場合は，ステータス・ワードのDONEビットだけでなくBUSYビットもセットして返します．このとき，次のキー入力を待っていてはいけません(NO WAIT)．

● STATUS(キャラクタ・デバイス)

コマンド・コード＝6, 10

13バイト	リクエスト・ヘッダ	(R/W)

このコマンドは，ドライバがデータ入力または出力を待っている状態かどうかの情報をMS-DOSに返します．ドライバは，その状態をステータス・ワードのBUSYビットで返します．

◆ 出力(コマンド・コード＝10)の場合

BUSYビット＝0で返すと，出力デバイスに対して直ちにライト・リクエストが発行されます．BUSYビット＝1を返すと，MS-DOSは現在のライト・リクエストが完了するまで待ちます．たとえば，通信ポート(RS-232C)がパラレル-シリアル変換中であるとか，プ

リンタがBUSYである場合などがこれに相当します．

◆ 入力(コマンド・コード=6)の場合

入力バッファにデータがあればBUSY=0，空であればBUSY=1を返します．

MS-DOSは，すべての入力デバイスに対してバッファリングの存在を想定しているので，これを行わないデバイスでは常にBUSY=0を返します．そうでないと，MS-DOSは存在しないバッファの入力を待ち続けることになります．

● FLUSH(キャラクタ・デバイス)

コマンド・コード=7，11

13バイト	リクエスト・ヘッダ	(R/W)

MS-DOSは，これらのコマンドによりドライバに対してすべてのデバイス・コマンドの打ち切りを指示します．

デバイス・ドライバは，これらのコマンドを受け付けたらキャラクタ・デバイスの入出力バッファを空にします．

● DEVICE OPEN/CLOSE

コマンド・コード=13，14

13バイト	リクエスト・ヘッダ	(R/W)

このコマンドは，ver.3.10以降のデバイス・ドライバで，デバイス属性のOPEN/CLOSE/RMビットがセットされている場合のみサポートされます．

このコマンドは，デバイス(ファイル)がオープンまたはクローズされるたびに呼び出されます．デバイス・ドライバは，これを利用してリファレンス・カウントを行うことができます．すなわち，現在のデバイス(ファイル)・オープンの数を知ることができるため，ブロック・デバイスではバッファの管理に使用することも可能です．

また，キャラクタ・デバイスの場合は，デバイス・オープンの開始を知ることができるため，たとえば，プリンタにおいてフォントの設定や制御文字列の処理を正確に行うことも可能になります．

● REMOVABLE MEDIA(ブロック・デバイス)

コマンド・コード=15

13バイト	リクエスト・ヘッダ	(R/W)

このコマンドは，ver.3.10以降のデバイス・ドライバで，デバイス属性のOPEN/CLOSE/RMビットがセットされている場合のみサポートされます．

このコマンドは，IOCTLシステム・コールのサブ・ファンクションによってブロック・デバイスのみでコールされます．

このコマンドは，ハード・ディスクなどのような交換不可能なメディアを扱うのか，あるいはフロッピ・ディスクのように交換可能なメディアを扱うのかを知るためにコールされます．デバイス・ドライバは，メディアの交換が可能な場合にステータス・ワードのBUSYビットを“0”にして返し，交換が不可能なメディアの場合には“1”にして返します．

● Generic IOCTL

コマンド・コード=19

13バイト	リクエスト・ヘッダ	(R/W)
1バイト	カテゴリ(メジャー)・コード	(R)
1バイト	ファンクション(マイナー)・コード	(R)
1ワード	SIの内容	(R)
1ワード	DIの内容	(R)
2ワード	バッファへのポインタ	(R)

このデバイス・コマンドは，ver.2.11におけるコマンド・コード3(IOCTL INPUT)およびコマンド・コード12(IOCTL OUTPUT)に代わるコマンドとして，ver.3.30以降において拡張されたデバイス・コマンドです．

これは，ver.3.30において拡張されたIOCTL機能(システム・コール)に対応するためのものです．新しいデバイス・ドライバは，このGeneric IOCTL機能を利用すべきです．

カテゴリ・コードやファンクション・コードは，ファンクション・リクエスト440CHや440DHから渡されます．他のフィールドに関しても，詳しいことは該当するファンクション・リクエストの解説を参照してください．

● Get/Set Logical Drive Map

コマンド・コード=23，24

13バイト	リクエスト・ヘッダ	(R/W)
1バイト	ユニット・コード	(R)
1バイト	デバイス・コード	(W)
1バイト	コマンド・コード	(R)
1ワード	ステータス	(R)
2ワード	予約	(R/W)

このコマンドは，デバイス属性のver.3.3ビットをセットしたデバイスのみに対して発行されます．また，このデバイス・コマンドは，拡張されたファンクション・リクエスト(IOCTLサブ・ファンクション)の実行によって，ブロック・デバイスのみに対して発行され

ます.

ユニット・コードとは，マップされた論理ドライブ名に対応した値です．デバイス・ドライバでは，デバイス・コードのフィールドに物理ドライブのオーナーである現在の論理ドライブを返します.

8-3 デバイス・ドライバのデバッグ方法

ここで，デバイス・ドライバのデバッグ方法について少し触れておきましょう．デバッグの方法としてはいくつかの方法が考えられますが，ここでは，二つの方法を示しておきます.

一つの方法は，目的のデバイス・ドライバをシステムに登録し，その登録したアドレスを探して，デバッガにより機械語レベルでデバッグする方法です．また一つは，デバイス・ドライバを一般のプログラムとして実行しながらデバッグを行う方法です.

● デバイス・ドライバを組み込んでデバッグする

この方法では，デバイスのINITコマンドは，システムのブート時に実行されるため，その時点で直接デバッガを使用することはできません.

(1) したがって，たとえばINITコマンド用のコマンド・パケットをデバッガの中で適当なエリアに作成し，ユーザがMS-DOSになったつもりでデバイス・ドライバのINITコマンドをリクエストする(エミュレートする).

この際に，エラーのシミュレーションも行っておいたほうがあとで楽になる.

(2) デバイス・ドライバをアクセスできるコマンドをデバッガによってロードする．ここでロードするコマンドはcommand.comであったり，場合によってはデバイスを直接アクセスするデモ用コマンドを作成することもある.

(3) デバッガを用いてデバイス・ドライバの絶対アドレスを探すか，デバイス・ドライバ(INITコマンド)内に自身のアドレス(CSレジスタの値)を知らせる機能を組み込むなどして，デバイス・ドライバの絶対アドレスを得る.

絶対アドレスが得られたら，デバイス・ドライバのアセンブル・リストなどを参照して，目的のデバイス・コマンドのオフセットを得る.

(4) これによって，目的の絶対アドレスを知ることができるので，そこにブレーク・ポイントを設定してデバイス・ドライバをアクセスする.

デバイス・ドライバのアクセスとは，すなわちデバッガでロードしているコマンドを実行することになる.

(5) その後は通常のプログラムと同様に，デバッガの機能をフルに活用してデバッグを行う.

● デバイス・ドライバをプログラムとしてデバッグする

デバイス・ドライバといえども，スタートアップ・ルーチンをリンクすれば，一般のプログラムと同様にデバッグすることが可能になります．すなわち，スタートアップ・ルーチンにデバイスの各コマンドをコールする機能を記述しておき，MS-DOSになったつもりでデバイス・ドライバを呼び出します(エミュレートする).

これによって，デバイス・ドライバもsymdebによるシンボリックなデバッグや，CodeViewを用いたソース・レベルでのデバッグが可能になります．ただし，この場合は一般のコマンド(command.comのdirなど)によるドライバのアクセスはできません.

このデバイス・ドライバをプログラムとしてデバッグする方法については，Appendix Bのグラフィック・コンソール・ドライバの実例のところで詳しく述べることにします.

* *

この章では，MS-DOSに対してユーザ・デバイスを拡張する際に必要となる知識として，デバイス・ドライバの構造を解説しました．デバイスを拡張する場合，CP/MなどではBIOSを改造するのが一般的で，当時の技術書ではこのBIOSの改造に関する記事が脚光をあびていました.

MS-DOSでは，CP/M路線からUNIX路線へと移行するとき，UNIXの特徴であるデバイス・ドライバの概念を取り入れました．これによってMS-DOSでは，デバイス・ドライバの拡張や変更が容易となり，この基本設計のよさが，少なからず日本語FEPなどの開発にも影響しているといえるでしょう.

このように，優れたデバイス・ドライバが開発されたことによって，その源であるMS-DOSの使い勝手がますます向上し，市場が拡大するにつれてソフトウェア開発に拍車がかかるといった，MS-DOSにとっては「良い方向への循環」が起こっています.

デバイス・ドライバは，一般のアプリケーションに比較してその制約も多く，また高級言語のみによって記述することができないため，多少むずかしく感じられるかもしれません．しかし，自分のデバイスをMS-DOSが認めてくれたときの感激もまたひとしおです.

読者諸兄諸姉も，自分なりのデバイス・ドライバを作成してみることによって，ソフトウェア開発のおもしろさが実感でき，大きな満足感を味わうことができるでしょう.

第9章 MS-DOSの拡張メモリ

EMSとマッピングとEMMファンクション

この章では，MS-DOS ver.3.30でサポートが開始された拡張メモリの詳細について解説します．まず，最初になぜ拡張メモリが必要なのか，その理由について考えます．次に，拡張メモリのアクセスに関する手順を概説したのち，拡張メモリの機能呼び出しの詳細について解説します．拡張メモリの応用例としては，Appendix CにRAMディスク・ドライバを実例として掲げてあります．

9-1 内部メモリの限界と拡張メモリ

MS-DOSも，これほどもてはやされ，漢字処理やグラフィックス処理の機能向上にともない，プログラム・サイズもしだいに肥大化の一途をたどっています．しかし，MS-DOSは8086 CPU用に設計されているため，8086 CPUのもつ1Mバイトのアドレス空間の中で機能しなければなりません．また，グラフィックV-RAMやROM領域さえも1Mバイト空間に押し込めているため，MS-DOSではシステムの管理するメモリ空間として640Kバイトが標準となっています．

さらにMS-DOSでは，子プロセスのコール機能をサポートしており，この機能はとても便利がゆえに多用すると内部のメモリ空間を圧迫する結果となってしまい，MS-DOSの640Kバイトの内部メモリでは限界を感じるようになってきました．

内部メモリの限界

ここで，MS-DOSの使用例としてプログラム開発を考えてみます．最近では，カナ漢字変換用に優れたフロント・エンド・プロセッサが出回り，これによってソース・プログラムのコメントに日本語を使う機会が多くなってきています．ソース・プログラムのコメントに日本語を使うと，プログラム開発者はもとより開発者以外の人がプログラムを読む場合に，正確にコメントが伝わるため誤解を受けることが少なくなります．

このため，ソース・プログラムの編集作業をはじめるまえに，この便利な小道具(日本語フロント・エンド・プロセッサ)をシステムにデバイス登録しておくのが常識的になってきました．この便利な小道具も機能の強化にともない，肥大化してしまって100Kバイト以上になってきました．したがって，MS-DOSのシステムが約100Kバイトで小道具が約100Kバイトとすると，MS-DOSの640Kバイトのうち，すでに200Kバイト強を使い切ってしまいます(図9-1)．

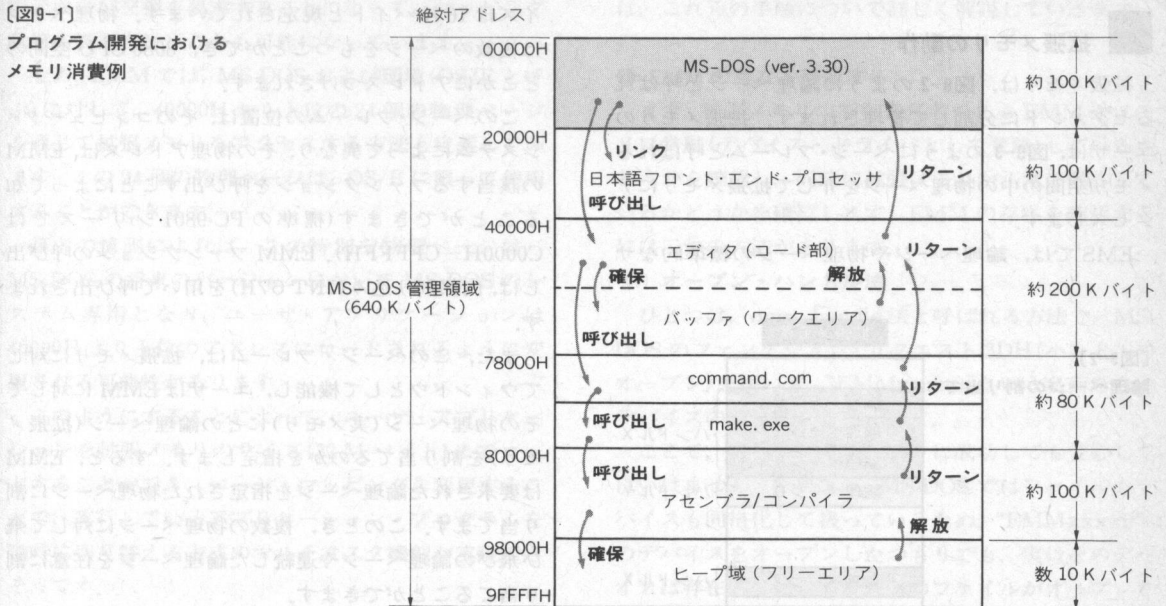
次にエディタです．エディタもまた高機能のフル・スクリーン・エディタが市販されていて，これも多量のメモリを消費しています．筆者の使っている市販のエディタは，プログラム自体が100Kバイト前後で，ワークエリア用に約100Kバイト前後もメモリを消費してしまいます．したがって，残りのメモリは250Kバイト程度になってしまいました．

さて，ソース・プログラムの編集を終えるとアセンブル/コンパイルの処理を行うことになりますが，ここで，いちいちエディタを終了してアセンブル/コンパイルを行うと，エラーが見つかったときに再び“エディタの起動→ソース・プログラムの該当する行の検索”などの煩わしい操作を繰り返さなければなりません．

このときに威力を発揮するのが，子プロセスのコール機能で，エディタの中からcommand.comを呼び出すことによって，そのつどエディタを終了することなくアセンブル/コンパイルを可能にしてくれます．いうまでもなく，このような処理方法は今や常識的な手段となっています．

すると，ここで250Kバイトのうちからcommand.comの常駐部分が差し引かれます．また，第1章でも述べたように，アセンブル/コンパイル作業においてはMAKEユーティリティを活用することが多く，さらに50Kバイト前後が消費される計算になります．

そして，最終的にコンパイラ(100Kバイト前後)が起動された時点では，残りのメモリが数10Kバイト程度しかなくなってしまいました．したがって，少し

〔図9-1〕
プログラム開発における
メモリ消費例

大きなソース・プログラムのアセンブル/コンパイルを実行しようとすると「メモリが足りない」旨のエラー・メッセージを表示して，無情にも処理をストップしてしまう結果となります．

このほかに高級言語のデバッグ段階では，CodeViewを利用することが多くなります．このCodeViewもまた他聞にもれずメモリの大喰者で，ことによってはターゲット・プログラムのロードさえもできない場合も生じてきます．

また，デバッグの途中でマップ情報をみたい場合などもあります．ここでも子プロセス・コール機能が威力を発揮し，CodeViewの中からcommand.comを呼び出したくなりますが，メモリの残り容量が少ないとCodeViewを終了せざるを得なくなり，デバッグ効率が著しく低下する結果となってしまいます．

以上の例からもわかるように，MS-DOSの640 Kバイトのメモリ空間は今や限界に達してきており，ユーザが快適な環境を望む場合に，なんらかの方法でメモリの拡張を行わなければならない時代が到来しているといえます．

拡張メモリ

これに対するMS-DOSの開発側(マイクロソフト社をはじめとする数社)からの答えのひとつが，拡張メモリ仕様EMS(Expanded Memory Specification)によるメモリの拡張方法です．

この拡張メモリは，MS-DOSの640 Kバイトの壁を破るためのひとつの道具であり，あとで詳しく述べるようにMS-DOSのマルチタスク化の手段となる可能性も秘めている興味深い道具です．

EMSは，拡張メモリに対するひとつのきまりを定めたもので，このEMS対応のメモリ・ボード(ハードウェア)であれば，どこのメーカから供給されているボードでも，アプリケーション側では同一の手順でアクセスすることが可能となっています．

これは，バンク切り替え方式のメモリ・ボードが，開発当初において各社まちまちの仕様だったのが，時間が経つにつれてボード・メーカによって統一されてきたのに比較し，当初からMS-DOSの開発者によって規格統一されているため，将来の互換性が保証されているもので，ユーザとしては安心して使用(購入)できるようになっています．

このEMS対応の拡張メモリへのアクセスは，拡張メモリ・マネージャEMM(Expanded Memory Manager)と呼ばれる拡張メモリの制御を行うソフトウェア(デバイス・ドライバ)を介して行われます．すなわち，拡張メモリを利用するには，EMSに対応したメモリ・ボード(ハードウェア)とEMM(ソフトウェア)の両者が揃ってはじめてアクセス可能となります．

EMSでは，最大32 Mバイトまでの拡張メモリの増設をサポートしています．8086 CPU(ファミリのリアル・モードも含む)では，1 Mバイトのメモリ空間しかもっていませんが，その1 Mバイト空間のうちのある範囲の物理アドレスをウィンドウとし，このウィンドウを通じて拡張メモリに対するアクセスを可能としています．

拡張メモリの動作

拡張メモリは，図9-2のように論理ページと呼ばれるセグメントに分割して管理されます．拡張メモリのユーザは，図9-3のようにページ・フレームと呼ばれるメモリ空間の中の物理ページを介して拡張メモリにアクセスします．

EMSでは，論理ページや物理ページの標準的なサイズは16Kバイトと規定されています．物理ページは複数のページをもつことができ，8086 CPU空間のどこかにアドレスづけされます．

このページ・フレームの位置は，そのコンピュータ・システムによって異なり，その物理アドレスは，EMMの該当するファンクションを呼び出すことによって知ることができます(標準のPC-9801シリーズではC0000H～CFFFFH)．EMMファンクションの呼び出しは，内部割り込み(INT 67H)を用いて呼び出されます．

また，このページ・フレームは，拡張メモリに対してウィンドウとして機能し，ユーザはEMMに対してその物理ページ(実メモリ)にどの論理ページ(拡張メモリ)を割り当てるのかを指定します．すると，EMMは要求された論理ページを指定された物理ページに割り当てます．このとき，複数の物理ページに対して飛び飛びの論理ページや連続した論理ページを任意に割り当てることができます．

ユーザは，物理ページに割り当てられた論理ページに対して，あたかも実メモリをアクセスするかのごとく自由に読み書きを行うことができます．拡張メモリは実アドレスに割り当てられるので，その拡張メモリ内にあらかじめ転送してあるプログラム(サブルーチンなど)の実行さえも可能になっています．

EMSでは，このように物理ページを割り当てることをマッピングと呼んでいて，EMMはそのための情報を内部ワーク・エリアにもっています．ユーザはEMMに対し，必要に応じてマッピング情報の保存や

〔図9-2〕
論理ページの割り当て

〔図9-3〕
拡張メモリへのアクセス

復元および交換を要求することによって，マッピング状態の変更を行うことも可能になっています．

なお，EMMでは，MS-DOSおよび環境(OS/Eと呼ぶ)に対して，40000Hから上位の24個の物理ページを通じて拡張メモリをアクセスする方法も定義しています．この24個の物理ページは，OS/Eに限って使用することができます．

筆者の憶測によれば，この特別な物理ページは，MS-DOSの将来のバージョンにおいてMS-DOSのシステム専用となり，ユーザ・アプリケーションは40000Hより上位のアドレスにロードされるように変更される可能性があります．

このようにすることによって，ユーザ・アプリケーションを拡張メモリのサイズ(32Mバイト)までロードすることができ，ページ・マッピングを変更することで，実行しているアプリケーション・プログラムを瞬時に切り替える方式のマルチタスク機能が実現できそうです．

ただし，この場合にユーザ・アプリケーションは，コードとデータを含めて384Kバイト以下でなければならず，もしそれ以上のメモリを必要とする場合には，アプリケーション自身がコードやデータを分割して拡張メモリを利用するようにプログラミングされている必要があります．

したがって，この方式によるMS-DOSの拡張は，拡張メモリの普及動向(市場人気)をにらみながら開発されていくものと思われ，現在の時点(ver.3.30)では「拡張メモリをサポートした」といっても，そのEMMはデバイス・ドライバとして利用可能となっているだけで，MS-DOS自身は何も関与していないようです．

拡張メモリの利用方法

拡張メモリを利用するには，次に示す手順にしたがい，EMMに対していろいろな処理の指定を行わなければなりません．

(1) EMMの存在の確認をする
(2) システムの環境を調べる
(3) 拡張メモリの割り当てを要求する
(4) 拡張メモリのマッピングを行う
(5) このとき，必要に応じてマッピング情報の保存や復元を行う
(6) 拡張メモリに対してデータの読み書きを行う．このとき，拡張メモリ上のプログラムに制御を移すことも可能
(7) アプリケーションを終了する際に，割り当てられている拡張メモリの返却を行う

以上が一般的な拡張メモリ利用手順となります．では，これらの手順について詳しく解説していきます．

● EMMの存在を確認する

まず，拡張メモリの制御責任者であるEMMがメモリに常駐(デバイス・ドライバとして登録)しているかどうかを確認し，同時に拡張メモリが正常に動作しているかどうかを確認します．EMMの存在を確認するには二つの方法があります．

(1) オープン・ハンドル法

ひとつは，open handle法と呼ばれる方法で，MS-DOSのファンクション・リクエスト3DH(ハンドルのオープン)を用いて，"EMMxxxx0"という名前をもつデバイスのオープンを行います．

ここで，デバイス・オープンに成功しても安心してはいけません．なぜなら，MS-DOSではファイルもデバイスも同格化して扱っているため，"EMMxxxx0"のデバイスをオープンしたつもりでも，実はそのデバイスは存在していなくて同名のファイルがオープンされている可能性もあるからです．

この相違を確認するために，ファンクション・リクエスト44H(デバイス制御)を用いてハンドルのIOCTLデータを調べます．このシステム・コールでDXレジスタのビット7がセットされて返されたら，そのハンドルはデバイスであることが確認でき，どうやらEMMは存在しているということになります．

次にファンクション・リクエスト4407H(出力ステータスの読み出し)を用いてデバイスのステータスを調べます．このとき，もしEMMが動作状態にあればALレジスタにFFHが返されるので，以降は安心してEMMにいろいろなファンクションを要求することが可能です．

これらのチェックに成功しても失敗しても，デバイス(ファイル)・オープンの際に返されたハンドルを，ファンクション・リクエスト3EH(ハンドルのクローズ)を用いてクローズしておきます．

(2) ゲット・インタラプト・ベクタ法

二つ目の方法は，get interrupt vector法と呼ばれる手法で，MS-DOSのファンクション・リクエスト35H(割り込みベクタの読み出し)を用いて割り込み番号67Hのベクタを取得します．

この割り込みベクタの指すセグメント・アドレスは，すなわちEMMのデバイス・ヘッダの先頭を指していることになり，そのオフセット0AHからの8バイトにはデバイス名が格納されているはずです．したがって，このデバイス・ヘッダに格納されているデバイス名が"EMMxxxx0"であるかどうかを調べることによってEMMの存在を確認できます．

ここで，通常のアプリケーション・プログラムであ

れば，EMMの確認方法としてopen handle法とget interrupt vector法のいずれの方法を使用しても問題ありません．しかし，プログラムがデバイス・ドライバであったり，ファイル・システムの操作中に割り込み処理をともなうものであるときは，必ずget interrupt vector法を使用しなければならないので注意が必要です．

● システムの環境を調べる

次に，アプリケーションはそのシステム固有のEMMの環境を調べる必要があります．

ここでは，まず拡張メモリのページがアプリケーションの必要とするサイズだけあるかどうかを確認します．このサイズの確認は，EMMファンクション03(未アロケート・ページ数の取得)を用いて行うことができます．

次に，拡張メモリをアクセスするためのページ・フレーム(ウィンドウ)に関する情報を取得しなければなりません．すなわち，ページ・フレームのアドレスや大きさなどを調べ，物理ページにアクセスする準備をしておく必要があります．

このページ・フレームのアドレスは，EMMファンクション02(ページ・フレームのアドレスの取得)を用いることによって知ることが可能です．また，ページ・フレーム内の物理ページの数は，EMMファンクション2501(物理ページ数の取得)を用いることによって取得できます．

前述したように，システムによっては物理ページが連続して配置されているとは限りません．このため，物理ページとそのセグメント・アドレスの対応を知るためにEMMファンクション2500(マップ可能な物理アドレス配列の取得)が用意されています．

EMMファンクション2500では，ES：DIレジスタが指すバッファに対し，セグメント・アドレスと物理ページ番号の対応表を物理ページの数だけ返してきます．

● 拡張メモリの要求

EMMの動作環境を把握したら，次に拡張メモリの割り当てを要求します．このためにEMMファンクション04(ページの割り当て)が用意されていて，このEMMファンクションに対して必要とするメモリ・サイズをページ数で要求します．この要求によって拡張メモリ領域が確保され，戻り値としてEMMハンドルが返されます．

EMMハンドルとは，ファイル・ハンドルと同様の概念で，EMMファンクション04で確保されたメモリ領域につけられるアクセス番号のことです．以後，その拡張メモリをアクセスするには，このEMMハンドル値を必要とするので，アプリケーション・プログラムでは，返されたEMMハンドル値をワーク域に保存しておく必要があります．

得られた拡張メモリ領域は論理ページ番号で管理され，論理ページの番号は0から順番につけられます．これはファイル管理における論理セクタ番号と同様の概念です．注意したいことは，拡張メモリの絶対アドレスが連続して論理ページに割り当てられるのではないということです．

論理ページのEMMハンドルに対する対応づけは，**図9-3**に示したように，そのときの拡張メモリの使用状態(拡張メモリの使用要求や返却の繰り返し)によって，飛び飛びのアドレスに論理ページの番号が割り付けられる可能性もあります．すなわち，拡張メモリへのアクセスは，“ハンドルXの論理ページのY番”という指定を行うことになります．

現在のバージョンのEMMでは，このときの拡張メモリの絶対アドレスに対して，どのようにEMMハンドルや論理ページが対応づけられているかを知ることはできません．

また，もし作業の途中で拡張メモリのサイズ変更が必要になった場合は，EMMファンクション18(ページの再割り当て)を用いることによって簡単にサイズの縮小や拡大が可能となっています．

● 論理ページのマッピング(対応づけ)

さて，拡張メモリにアクセスするには，その拡張メモリの論理ページを物理ページにマッピングして，実際の読み書きのアクセスは物理ページに対して行います．

この際に，物理ページへの論理ページのマッピングは，EMMファンクション05(ハンドル・ページのマップ/アンマップ)を呼び出すことによって可能となります．

ただし，このEMMファンクション05は，一度に1ページずつしかマッピングすることができません．複数のページをマッピングする必要がある場合(アクセスが16Kバイトを越える場合)は，このEMMファンクション05を繰り返すか，物理ページと論理ページの対応表を作成し，EMMファンクション1700(複数ページのマップ/アンマップ)を呼び出すことによって可能となります．

ここで，EMMファンクション1700では，その対応を物理ページの番号によって行いますが，もし物理ページのセグメント・アドレスを使用したい場合は，EMMファンクション1701を使用することによって可能となっています．物理ページに論理ページがマッ

ピングされたら，データの読み書きは物理ページ(標準メモリ・アドレス)に対して自由に行うことができます．

このとき，拡張メモリはデータだけでなくプログラム領域として使用することもでき，そのためのEMMファンクションも用意されています．拡張メモリ上のプログラムに制御を移すには，FAR JMPとFAR CALLの2種類が考えられますが，EMMファンクションでも，FAR JMPに対応するものとFAR CALLに対応するものが用意されています．

拡張メモリ内のプログラムにJMPするには，次の手順が必要となります．

① 物理ページに対して目的のプログラムが存在する論理ページをマッピングする．
② その物理ページ内の目的のプログラムにFAR JMPする．

EMMでは，ファンクション22(ページ・マップの変更とジャンプ)がこれらの処理をまとめて実行してくれます．この際にEMMファンクション22では，複数のページを割り当てることが可能なため，標準のPC-9801シリーズでは，最大64Kバイトまでのプログラムを実行することが可能です．

また，拡張メモリ内のプログラムをFAR CALLする場合は，制御がもとのプログラムに戻ってくるためにFAR JMPの場合に比較して次のように手順が増えます．

① 物理ページに対して目的のプログラムが存在する論理ページをマッピングする．
② その物理ページ内の目的のプログラムをFAR CALLする．
③ 物理ページに対する論理ページのマッピングをもとの状態に戻す．
④ 呼び出したプログラムに制御を戻す．

EMMでは，ファンクション23(ページ・マップの変更とコール)を用いることによって，これらの処理をまとめて実行してくれます．ここでもファンクション23では，複数ページのマッピングが可能なため，標準のPC-9801シリーズでは最大64Kバイトのサブルーチンを実行可能となっています．

さて，これらの拡張メモリをアクセスする際に，ある論理ページを複数の物理ページにマッピングした場合，たとえば物理ページ番号0と1に論理ページ10を重複してマッピングした場合はどうなるのでしょうか．

これは，その拡張メモリ・システムによって差が出てくるので注意が必要です．物理ページに対する論理ページのマッピングがハードウェアで行われているのであれば，物理ページにはまったく同一の論理ページが現れ，物理ページ0に書いたデータを物理ページ1で読みとることも可能です．

これに対して，マッピングがソフトウェアでエミュレーションされている場合は，物理ページ0に書いたデータを物理ページ1で読みとることはできません．

したがって，もしも重複したマッピングを必要とするアプリケーションの場合は，その拡張メモリ・システムがどちらの方法を採用しているのかを事前に調査しておく必要があります．

● マッピング情報の保存と復元

ここで，プログラムが通常のアプリケーションでなく，デバイス・ドライバであったりファイル・オープンの際に割り込みをともなう場合や，メモリに常駐するプログラムなどの場合は，マッピング情報の保存や復元の操作が必要となります．

たとえば，RAMディスク(デバイス・ドライバ)の場合に，あるアプリケーション(ユーザ・プログラム)で拡張メモリ(論理ページ)を物理ページにマッピングしてアクセスしているときに，ファイル・オープンなどの目的でそのRAMディスクにアクセスしたとします．

このときにRAMディスク・ドライバ側では，ドライバ自身でも拡張メモリを物理ページにマッピングして所定の仕事をすることになりますが，その状態のままアプリケーションに処理を返したとすると，アプリケーション側ではRAMディスク・ドライバでマッピングした論理ページをアクセスしてしまうことになります．

このため，デバイス・ドライバでは，拡張メモリをマッピングするまえに，以前のマッピング状態をEMMファンクション08(指定したページ・マップの保存)を用いて保存します．ここで，EMMファンクション08では，保存すべきマッピング状態に対応したEMMハンドルを必要とします．

デバイス・ドライバなどでEMMハンドル値が不明な場合は，EMMファンクション1500(すべてのページ・マップの保存)を用いることによって，すべてのマッピング状態の保存を行うことが可能となっています．また，複数の物理ページのうち一部の物理ページのマッピング状態を保存したい場合には，EMMファンクション1600(一部のページ・マップの保存)が有効です．

これらのファンクションを用いてマッピング状態が保存されたら，ドライバ側では拡張メモリを自由に操作できることになります．

物理ページと論理ページのマッピング状態をもとの状態に戻すには，ファンクション09(指定したページ・マップの復元)を用います．しかし，このファンクショ

ン 09 でも EMM ハンドルを必要とするため，デバイス・ドライバなどでその EMM ハンドル値が不明な場合は，ファンクション 1501(すべてのページ・マップの復元)を用います.

また，一部のマッピング情報の復元を行うのであればファンクション 1601(一部のページ・マップの復元)が有効です.

● **拡張メモリの返却**

あるアプリケーションで，拡張メモリを割り当てられたままそのプログラムを終了すると，EMM はその拡張メモリが不要になったかどうかを判断できないため，その不要になっている拡張メモリ領域をほかのアプリケーションに割り当てることができません.

このため，アプリケーションでは，EMM ファンクション 04(ページの割り当て)で要求したメモリ領域を EMM ファンクション 06(ページの解放)を用いて EMM に返却します.これによって，EMM はその不要になった拡張メモリ領域をほかのアプリケーションに割り当てることが可能となります.

9-2 EMM ファンクション

ここでは，拡張メモリを利用するために必要となる EMM ファンクションの詳細について解説します.

表9-1 は，EMM ファンクションの一覧表です. EMM ファンクションでは，AH レジスタに機能コードを設定し，ほかのレジスタにも必要なパラメータを設定します. そして，内部割り込み(INT 67H)を実行することにより EMM ファンクションが処理され，その結果が各レジスタおよびバッファに対して返されます.

〔表9-1〕 EMM ファンクション一覧 ①

番号	ファンクション名	機能	コール	リターン
01	Get Status	ステータスの取得	AH=40H	AH ← ステータス・コード
02	Get Page Frame Address	ページ・フレームのアドレス取得	AH=41H	AH ← ステータス・コード BX ← ページ・フレームのセグメント・アドレス
03	Get Unallocated Page Count	未アロケート・ページ数の取得	AH=42H	AH ← ステータス・コード BX ← 未アロケート・ページ数 DX ← 総ページ数
04	Allocate Pages	ページの割り当て	AH=43H BX=アロケートしたいページ数	AH ← ステータス・コード DX ← EMM ハンドル
05	Map/Unmap Handle Page	ハンドル・ページのマップ/アンマップ	AH=44H AL ← 物理ページ番号 BX ← 論理ページ番号 DX ← EMM ハンドル	AH ← ステータス・コード
06	Deallocate Pages	ページの解放	AH=45H DX ← EMM ハンドル	AH ← ステータス・コード
07	Get Version	バージョン番号の取得	AH=46H	AH ← ステータス・コード AL ← バージョン番号
08	Save Page Map	ページ・マップの保存	AH=47H DX ← EMM ハンドル	AH ← ステータス・コード
09	Restore Page Map	ページ・マップの復元	AH=48H DX ← EMM ハンドル	AH ← ステータス・コード
12	Get Handle Count	ハンドル・カウントの取得	AH=4BH	AH ← ステータス・コード BX ← オープンしている EMM ハンドルの数
13	Get Handle Pages	ハンドル・ページの取得	AH=4CH DX ← EMM ハンドル	AH ← ステータス・コード BX ← 指定された EMM ハンドルに割り当てられている論理ページ数

〔表9-1〕EMMファンクション一覧②

番号	ファンクション名	機能	コール	リターン
14	Get All Handle Pages	全ハンドル・ページの取得	AH=4DH ES:DI←オープンされているすべてのEMMハンドルのコピーと，各ハンドルに割り当てられているページ数が格納されるバッファへのポインタ	AH←ステータス・コード BX←オープンされているEMMハンドルの総数
15 コード 00H	Get Page Map	ページ・マップの保存	AX=4E00H ES:DI←バッファへのポインタ	AH←ステータス・コード バッファ←マッピング・レジスタの状態
15 コード 01H	Set Page Map	ページ・マップの復元	AX=4E01H ES:DI←バッファへのポインタ	AH←ステータス・コード
15 コード 02H	Get & Set Page Map	ページ・マップの取得と設定	AX=4E02H ES:DI←マップ・レジスタを保存するバッファへのポインタ DS:SI←マップ・レジスタに設定する配列(バッファ)へのポインタ	AH←ステータス・コード バッファ←マッピング・レジスタの状態
15 コード 03H	Get Size of Page Map Save Array	ページ・マップを格納するための配列サイズの取得	AX=4E03H	AH←ステータス・コード AL←配列サイズ(バイト数)
16 コード 00H	Get Partial Page Map	一部のマッピング情報の保存	AX=4F00H DS:SI←マップの一部を指定するデータへのポインタ ES:DI←バッファへのポインタ	AH←ステータス・コード バッファ←マップ・テキスト
16 コード 01H	Set Partial Page Map	一部のマッピング情報の復元	AX=4F01H DS:SI←バッファへのポインタ	AH←ステータス・コード
16 コード 02H	Get Size of Partial Page Map Save Array	一部のマッピング情報を格納する配列サイズの取得	AX=4F02H BX←部分的にマップされるページ数	AH←ステータス・コード AL←配列サイズ(バイト数)
17 コード 00H	Map/Unmap Multiple Handle Pages (Logical Page/Physical Page Method)	複数ページのマップ/物理ページの解放 (論理ページ/物理ページ方式)	AX=5000H DX←EMMハンドル CX←配列内のエントリ数 DS:SI←配列構造へのポインタ	AH←ステータス・コード
17 コード 01H	Map/Unmap Multiple Handle Pages (Logical Page/Segment Address Method)	複数ページのマップ/物理ページの解放 (論理ページ/セグメント・アドレス方式)	AX=5001H DX←EMMハンドル CX←配列内のエントリ数 DS:SI←配列構造へのポインタ	AH←ステータス・コード
18	Reallocate Pages	ページの再割り当て	AH←51H DX←EMMハンドル BX←再割り当てのページ数	AH←ステータス・コード BX←再割り当てされたページ数
19 コード 00H	Get Handle Attribute	ハンドル属性の取得	AX=5200H DX←EMMハンドル	AH←ステータス・コード AL←ハンドルの属性
19 コード 01H	Set Handle Attribute	ハンドル属性の設定	AX=5201H DX←EMMハンドル BL←ハンドルの新しい属性	AH←ステータス・コード
19 コード 02H	Get Attribute Capability	不揮発性属性のサポート可能性の調査	AX=5202H	AH←ステータス・コード AL←属性の性能
20 コード 00H	Get Handle Name	ハンドル名の取得	AX=5300H DX←EMMハンドル ES:DI←バッファ(8バイト)へのポインタ	AH←ステータス・コード バッファ←ハンドル名

〔表9-1〕EMMファンクション一覧 ③

番号	ファンクション名	機能	コール	リターン
20 コード 01H	Set Handle Name	ハンドル名の設定	AX=5301H DX←EMMハンドル ES : DI←ハンドル名へのポインタ	AH←ステータス・コード
21 コード 00H	Get Handle Directory	ハンドルのディレクトリ情報の取得	AX=5400H ES : DI←バッファへのポインタ	AH←ステータス・コード AL←エントリ数(ハンドル数) バッファ←ディレクトリ情報
21 コード 01H	Search for Named Handle	指定の名前をもつハンドルの検索	AX=5401H DS : SI←ハンドル名へのポインタ	AH←ステータス・コード DX←検索されたEMMハンドル
21 コード 02H	Get Total Handles	ハンドルの総数の取得	AX=5402H	AH←ステータス・コード BX←ハンドル総数
22	Alter Page Map & Jump	ページ・マップの変更とジャンプ	AH=55H AL←モード(0 : ページ番号, 1 : セグメント) DX←EMMハンドル DS : SI←ジャンプ・アドレスを含むデータ構造へのポインタ	AH←ステータス・コード
23	Alter Page Map & Call	ページ・マップの変更とコール	AH=56H AL←モード(0 : ページ番号, 1 : セグメント) DX←EMMハンドル DS : SI←ターゲット・アドレスを含むデータ構造へのポインタ	AH←ステータス・コード
23 コード 02H	Get Page Map Stack Space Size	ページ・マップの変更に必要なスタック・サイズの取得	AX=5602H	AH←ステータス・コード BX←必要とするスタック・サイズ(バイト数)
24 コード 00H	Move Memory Region	メモリ領域の移動	AX=5700H DS : SI←ソースとデスティネーション・アドレスの情報を含むデータ構造へのポインタ	AH←ステータス・コード
24 コード 01H	Exchange Memory Region	メモリ領域の交換	AX=5701H DS : SI←ソースとデスティネーション・アドレスの情報を含むデータ構造へのポインタ	AH←ステータス・コード
25 コード 00H	Get Mappable Physical Address Array	マップ可能な物理アドレス配列の取得	AX=5800H ES : DI←バッファへのポインタ	AH←ステータス・コード CX←物理ページのエントリ数
25 コード 01H	Get Mappable Physical Address Array Entries	マップ可能な物理アドレス配列のエントリ数の取得	AX=5801H	AH←ステータス・コード CX←物理ページのエントリ数
26* コード 00H	Get Hardware Configuration Array	ハードウェア構成に関する情報の取得	AX=5900H ES : DI←バッファへのポインタ	AH←ステータス・コード バッファ←拡張メモリ・ハードウェアの情報
26* コード 01H	Get Unallocated Raw Page Count	未アロケートのロウ・ページ・カウントの取得	AX=5901H	AH←ステータス・コード BX←未アロケートのロウ・ページ数 DX←ロウ・ページの総数
27 コード 00H	Allocates Standard Pages	標準サイズのページの割り当て	AX=5A00H BX←要求する標準ページ数	AH←ステータス・コード DX←EMMハンドル

〔表9-1〕EMMファンクション一覧 ④

番号	ファンクション名	機　　能	コ　ー　ル	リ　タ　ー　ン
27 コード 01H	Allocates Raw Pages	ロウ・ページの割り当て	AX=5A01H BX←要求するロウ・ページ	AH←ステータス・コード DX←EMMハンドル
28* コード 00H	Get Alternate Map Register Set	代替マップ・レジスタ・セットの取得	AX=5B00H	AH←ステータス・コード BL←マップ・レジスタ・セットの番号 ES：DI←マップ・レジスタ・コンテキストのセーブ・エリアへのポインタ
28* コード 01H	Set Alternate Map Register Set	代替マップ・レジスタ・セットの設定	AX=5B01H BL←マップ・レジスタ・セットの番号 ES：DI←マッピング・レジスタ・コンテキストのリストア・エリアへのポインタ	AH←ステータス・コード
28* コード 02H	Get Alternate Map Save Array Size	代替マップ・セーブ配列のサイズ取得	AX=5B02H	AH←ステータス・コード DX←配列のバイト数
28* コード 03H	Allocate Alternate Map Register Set	代替マップ・レジスタ・セットの割り当て	AX=5B03H	AH←ステータス・コード BL←マップ・レジスタ・セットの番号
28* コード 04H	Deallocate Alternate Map Register Set	代替マップ・レジスタ・セットの解放	AX=5B04H BL←マップ・レジスタ・セットの番号	AH←ステータス・コード
28* コード 05H	Allocate DMA Register Set	DMAレジスタ・セットの割り当て	AX=5B05H	AH←ステータス・コード BL←DMAレジスタ・セットの番号
28* コード 06H	Enable DMA on Alternate Map Register Set	代替マップ・レジスタによるDMAの使用許可	AX=5B06H BL←DMAレジスタ・セットの番号 DL←DMAチャネル番号	AH←ステータス・コード
28* コード 07H	Disable DMA on Alternate Map Register Set	代替マップ・レジスタに対応するDMAの使用禁止	AX=5B07H BL←マップ・レジスタ・セット番号	AH←ステータス・コード
28* コード 08H	Deallocate DMA Register Set	DMAレジスタ・セットの解放	AX=5B08H	AH←ステータス・コード BL←DMAレジスタ・セットの番号
29	Prepare Expanded Memory Hardware for Warm Boot	ウォーム・ブートのための拡張メモリ・ハードウェアの準備	AH=5CH	AH←ステータス・コード
30* コード 00H	Enable OS/E Function Set	OS/Eファンクション・セットの使用許可	AX=5D00H BX，CX←アクセス・キー	AH←ステータス・コード BX，CX←アクセス・キー
30* コード 01H	Disable OS/E Function Set	OS/Eファンクション・セットの使用禁止	AX=5D01H BX，CX←アクセス・キー	AH←ステータス・コード BX，CX←アクセス・キー
30* コード 02H	Return Access Key	アクセス・キーのリターン	AX=5D02H BX，CX←アクセス・キー	AH←ステータス・コード
31 コード 00H	Get Page Frame Status	ページ・フレーム用バンクのステータス取得	AX=7000H	AX←ステータス・コード AL←ステータス （00H：ページ・フレーム使用可 01H：ページ・フレーム使用不可）
31 コード 01H	Enable/Disable Page Frame	ページ・フレーム用バンクのステータス設定	AX=7001H BL←指示コード （00H：ページ・フレームに使用 01H：VRAMに使用）	AH←ステータス・コード AL←ステータス （00H：ページ・フレームに使用可 01H：ページ・フレームに使用不可）

（注）＊印はOSのみが使用可能なファンクション

また，EMMファンクションの実行でエラーがあったかどうかは，AHレジスタに返されるEMMステータス・コード(**表9-2**)によって知ることができます．

Get Status	No.01
機能	ステータスの取得
コール	AH＝40H
リターン	AH←ステータス・コード

メモリ・マネージャが存在し，かつハードウェアが正常に動作しているかどうかのチェックを行います．その結果は，AHレジスタにステータス・コードとして返されるので，そのステータス・コードによって判断します．

Get Page Frame Address	No.02
機能	ページ・フレーム・アドレスの取得
コール	AH＝41H
リターン	AH←ステータス・コード BX←ページ・フレーム・セグメント・アドレス(AH＝00Hの場合のみ)

ページ・フレームが割り当てられているセグメント・アドレスを返します．

Get Unallocated Page Count	No.03
機能	未アロケート・ページ・カウントの取得
コール	AH＝42H
リターン	AH←ステータス・コード BX←未アロケート・ページ数 DX←総ページ数

〔表9-2〕拡張メモリ・マネージャのステータス・コード

ステータス・コード	内容
00H	正常実行
80H	拡張メモリの管理プログラムが動作しない
81H	拡張メモリのハードウェアが動作しない
82H	バージョン3.2以下で使用される（busyステータス）
83H	指定のEMMハンドルが見つからない
84H	ファンクション・コードが未定義
85H	すべてのEMMハンドルが使用されている
86H	指定のEMMハンドル用のセーブ・エリア内にページ・マッピング・レジスタの情報が含まれている
87H	要求されたページがシステムに存在しない
88H	要求された未アロケートのページがない
89H	現在のバージョンでは使用されない(ハンドルにゼロ・ページを割り当てられない)
8AH	メモリにマップする論理ページがハンドルに割り当てられている論理ページの範囲外にある
8BH	指定された物理ページがマップできない
8CH	ページ・マッピング・コンテキストをストアするエリアが一杯になった
8DH	EMMハンドルで指定したページ・マッピング・コンテキストがすでにセーブされている
8EH	EMMハンドルで指定したページ・マッピング・コンテキストが存在しない
8FH	サブ・ファンクションのパラメータ(ALレジスタ)が定義されていない
90H	指定の属性は定義されていない
91H	システム構成が不揮発性をサポートしていない
92H	拡張メモリのソース領域とコピー先の領域が同一のハンドルをもちオーバラップしている(一部がオーバライトされた)
93H	指定された拡張メモリのソース領域かコピー先の領域が相手の領域(ページ)より大きい
94H	標準メモリと拡張メモリの領域がオーバラップしている
95H	論理ページ内のオフセットが論理ページの大きさを越えた
96H	領域の大きさが1Mバイトを越えた
97H	拡張メモリのソース領域と交換先の領域が同一のハンドルをもちオーバラップしている(無効)
98H	ソースとデスティネーションのメモリのタイプが未定義(またはサポートされていない)
9AH	指定の代替用マップ・レジスタがサポートされていない
9BH	すべての代替用マップ・レジスタ(DMAレジスタ・セット)が割り当てられている
9CH	代替用マップ・レジスタ(DMAレジスタ・セット)がサポートされていない
9DH	指定された代替用マップ・レジスタ(DMAレジスタ・セット)が定義されていないか，割り当てられていないか，または現在割り当て済みのマップ・レジスタ・セットである
9EH	専用のDMAチャネルがサポートされていない
9FH	指定したDMAチャネルはサポートされていない
A0H	指定されたハンドル名に対するハンドル値が見つからない
A1H	指定されたハンドル名はすべて存在している
A2H	コピー/交換中に1Mバイトのアドレス空間を越えようとした
A3H	ファンクションに渡されたデータ構造が不正である
A4H	OSがこのファンクションのアクセスを拒否した

拡張メモリの総ページ数と，いまだに割り当てられていないページ数を返します．ユーザは，このファンクションによって必要とするメモリ容量が，拡張メモリに残っているかどうかを判断します．

Allocate Pages	No.04
機能	ページの割り当て
コール	AH＝43H
	BX←要求するページ数
リターン	AH←ステータス・コード
	DX←EMMハンドル

BXレジスタによって要求された拡張メモリのページ数の割り当てを行います．

ページの割り当てに成功すると，EMM固有のハンドル(1～255の値)が返されるので，ユーザは以後の拡張メモリをアクセスする際には，このEMMハンドルを使用します．すなわち，EMMハンドルはファイル・ハンドルと同様の考えかたで使用します．

なお，このファンクションで扱うページのサイズは，標準サイズ(16Kバイト)になっていて，ハンドルに0ページを割り当てることはできません．0ページを割り当てたい場合はファンクション27(ページの割り当て)を使用します．

また，ハンドルの0000Hは，OSのみが使用可能なハンドルとして準備されていますが，このファンクション04ではハンドル0000Hを扱うことができません．OSは，0000Hのハンドルを用いることによって，どのEMMファンクションでも呼び出すことが可能になっています．

そして，この0000Hのハンドルにページを割り当てるにはファンクション18(ページの再割り当て)を用います．この0000Hの特別なハンドルは，アプリケーションでは扱うことができないので注意が必要です．

Map/Unmap Handle Page	No.05
機能	ハンドル・ページのマップ/アンマップ
コール	AH＝44H
	AL←物理ページ番号
	BX←論理ページ番号
	DX←EMMハンドル
リターン	AH←ステータス・コード

このファンクションは，ALレジスタで指定された物理ページ(PC-9801シリーズでは0～3ページ)に対してBXレジスタで指定された論理ページをマッピングします．このときに，指定の物理ページがどのセグメントに対応しているかを調べるには，ファンクション25やファンクション02を用いて調べることができます(代表的なPC-9801シリーズでは，先頭のセグメント・アドレスC000H～CC00Hとなっている)．

また，このファンクションでは，ALレジスタで指定した物理ページに対して，BXレジスタでFFFFHの論理ページを指定することによって，その物理ページの読み書きを不可能にすることもできます(物理ページの解除)．これによって，その物理ページの内容を完全に保存することができます(誤ってメモリ内容を破壊されることがない)．

ここで，ページの解除を行う際には，そのまえにマッピング情報の保存をしておかなければなりません．マッピング情報の保存には，ファンクション8,15,16を用います．また，保存したマッピング情報の復帰を行うにはファンクション9,15,16を用います．

Deallocate Pages	No.06
機能	ページの解放
コール	AH＝45H
	DX←EMMハンドル
リターン	AH←ステータス・コード

このファンクションは，EMMハンドルに割り当てられている物理ページの解放を行います．アプリケーション・プログラムは，OSに戻るまえにEMMハンドルに割り当てているページを，このファンクションによって解放しなければなりません．

そうすることによって，ほかのアプリケーション・プログラムがそのページを利用できるようになります．また，このファンクションによってページの解放が行われると，ハンドル名はすべてNULLに設定されます．

Get Version	No.07
機能	バージョンの取得
コール	AH＝46H
リターン	AH←ステータス・コード
	AL←バージョン番号

このファンクションは，メモリ・マネージャのバージョン番号をALレジスタに返します．バージョン番号は，図9-4のようにBCD形式で返され，上位4ビットがバージョン番号の整数部分を，下位4ビットが小数部分を表します．

〔図9-4〕
ALレジスタに返されるバージョン番号

Save Page Map	No.08
機能	ページ・マップのセーブ
コール	AH＝47H DX←EMMハンドル
リターン	AH←ステータス・コード

このファンクションは，拡張メモリ・ボード上のマッピング情報を内部エリアに格納します．

拡張メモリを使用している常駐プログラムや割り込み処理ルーチン，およびデバイス・ドライバは，割り込みやMS-DOSからの呼び出しを受けた場合に，ほかのアプリケーション・プログラムが拡張メモリを使用している可能性があるので，マッピング情報を保存しなければなりません．

なお，このファンクションはEMS ver.3.xとの互換性を考慮したファンクションであり，EMMハンドルを必要とすることや，64Kバイトのページ・フレームのみに関するマッピング情報だけを保存するため，EMS ver.4.x以降ではファンクション15や16を用います．

Restore Page Map	No.09
機能	ページ・マップの復帰
コール	AH＝48H DX←EMMハンドル
リターン	AH←ステータス・コード

このファンクションは，ファンクション08で格納されたマッピング情報の復帰を行います．

このファンクションも，EMS ver.3.xとの互換性のために用意されているもので，EMS ver.4.x以降の場合は，ファンクション15または16を用います．

Get Handle Count	No.12
機能	ハンドル数の取得
コール	AH＝4BH
リターン	AH←ステータス・コード BX←オープンしているEMMハンドル数

このファンクションでは，オープンされているEMMハンドルの数(OS専用のハンドル0000Hも含む)をBXレジスタに返します．

Get Handle Pages	No.13
機能	ハンドル・ページの取得
コール	AH＝4CH DX←EMMハンドル
リターン	AH←ステータス・コード BX←指定のEMMハンドルに割り当てられている論理ページ数

このファンクションでは，DXレジスタで指定されたEMMハンドルに割り当てられている論理ページ数をBXレジスタに返します．

Get All Handle Pages	No.14
機能	全ハンドル・ページの取得
コール	AH＝4DH ES：DI←バッファへのポインタ
リターン	AH←ステータス・コード BX←オープンしているEMMハンドルの総数(OSのハンドル0000Hを含む)

このファンクションでは，オープンされているEMMハンドルの数(OS専用のハンドル0000Hを含む)をBXレジスタに，また各ハンドルに割り当てられているページ数の情報をES：DIレジスタで指定されたバッファに返します．

このとき，バッファに返される割り当て情報の内容は，図9-5に示すように最初の1ワードにEMMハンドルの値が入り，次の1ワードに，そのEMMハンドルに割り当てられているページ数が入ります．ここで，バッファのサイズとして，

　オープンされているEMMハンドルの数
×2ワード

だけの十分な領域を確保しておかなければなりません．バッファ・サイズの計算には，ファンクション12(オープンされているEMMハンドル数の取得)などを用います．

Get Page Map	No.15 コード 00H
機能	ページ・マップの取得
コール	AX＝4E00H ES：DI←バッファへのポインタ
リターン	AH←ステータス・コード

このファンクションでは，すべてのマッピング情報をES：DIレジスタで指定されたバッファに格納します．

このファンクションでは，ファンクション08と異なりEMMハンドルを必要としません．ES：DIレジスタで指定するバッファのサイズは，ファンクション1503を用いることによって決定することができます．

Set Page Map	No.15 コード 01H
機能	ページ・マップの設定
コール	AX＝4E01H DS：SI←バッファへのポインタ
リターン	AH←ステータス・コード

このファンクションでは，DS：SIレジスタで指定

したバッファのマッピング情報(ファンクション1500で格納)を復帰し，もとのマッピング状態に戻します．

このファンクションは，ファンクション09の代用として使用することができます．また，このファンクションでは，EMMハンドルを必要としません．

Get & Set Page Map	No.15 コード 02H
機能	ページ・マップの取得と設定
コール	AX＝4E02H DS：SI←ソース・バッファへのポインタ ES：DI←デスティネーション・バッファへのポインタ
リターン	AH←ステータス・コード

このファンクションでは，マッピング情報の格納と復帰を一度に行うことができます．このファンクションは，たとえば，拡張メモリの現在のマッピングを保存し，指定したマッピング状態にしたい場合などに使用します．

このファンクションでは，まずES：DIレジスタで指定したバッファに現在のマッピング情報を格納し，次にDS：SIレジスタで指定されたバッファの内容にしたがって拡張メモリのマッピングを行います．

Get Size of Page Map Save Array	No.15 コード 03H
機能	ページ・マップ格納配列のサイズ取得
コール	AX＝4E03H
リターン	AH←ステータス・コード AL←配列のサイズ(バイト数)

このファンクションでは，前述のファンクション1500～1502で必要とするマッピング情報格納バッファのサイズ(バイト数)をALレジスタに返します．

ページ・マップの格納サイズ(配列)は，拡張メモリのシステム構成や，マネージャの動作に依存していて一義的に規定されてはいません．

Get Partial Page Map	No.16 コード 00H
機能	一部のマッピング情報の格納
コール	AX＝4F00H DS：SI←マップの一部を指定するデータへのポインタ ES：DI←バッファへのポインタ
リターン	AH←ステータス・コード

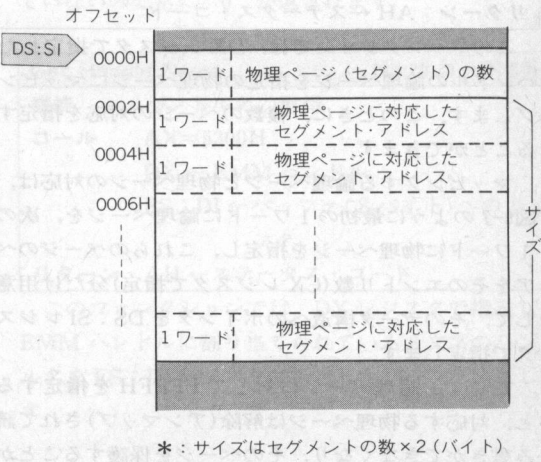
このファンクションは，DS：SIレジスタが指すデータ構造で指定された一部の物理ページのマッピング情報を，ES：DIレジスタで指定されたバッファに格納します．

このファンクションでは，一部のマッピング情報だけを扱うので，ファンクション15に比較してマッピング情報の格納に使用するメモリが小さくでき，同時に処理速度も速くなります．

一部の物理ページ(DS：SIレジスタで指定)は，**図9-6**のデータ構造で指定し，最初の1ワードで物理ページの数を指定し，次に物理ページに対応したセグメント・アドレスを指定します．このとき，セグメント・アドレスはマップ可能なセグメントでなければならず，どのセグメントがマップ可能かはファンクション25を用いて調べることができます．

また，ES：DIレジスタで指定するバッファのサイズは，ファンクション1602を使用することによって知ることができます．

〔図9-5〕ファンクション14で返されるページの割り当て情報

〔図9-6〕ファンクション1600で指定する一部のページの指定方法

Set Partial Page Map	No.16 コード 01H
機能	一部のマッピング情報の復帰
コール	AX＝4F01H
	DS：SI ←バッファへのポインタ
リターン	AH←ステータス・コード

このファンクションでは，DS：SIレジスタで指定した一部のマッピング情報(ファンクション1600で格納)を復帰してマッピング状態をもとの状態に戻します．

Get Size of Partial Page Map Save Array No.16 コード 02H	
機能	一部のマッピング情報を格納する配列のサイズ取得
コール	AX＝4F02H
	BX←部分的にマップされるページ数
リターン	AH←ステータス・コード
	AL←配列のサイズ(バイト数)

このファンクションは，ファンクション1600および1601でマッピング情報の格納に使用するメモリ領域のサイズ(バイト数)をALレジスタに返します．

ここで，ページ・マップの格納サイズ(配列)は，拡張メモリのシステム構成や，マネージャの動作に依存していて一義的には規定されていません．

Map/Unmap Multiple Handle Pages (Logical Page/Physical Page Method) No.17 コード 00H	
機能	複数ページのマップ/アンマップ(論理ページ/物理ページ方式)
コール	AX＝5000H
	DX←EMMハンドル
	CX←配列内のエントリ数
	DS：SI←配列構造へのポインタ
リターン	AH←ステータス・コード

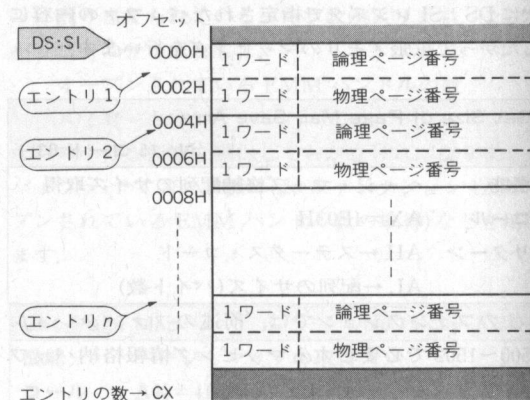
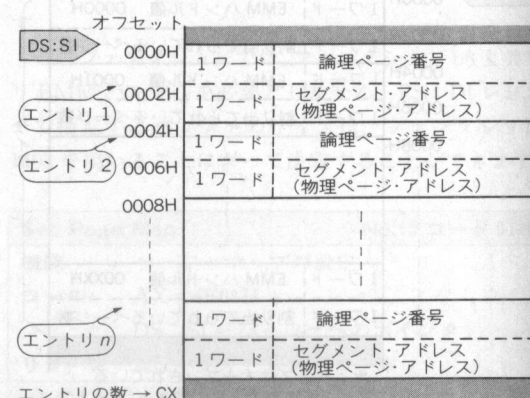
このファンクションでは，DXレジスタで指定したハンドルの論理ページを指定の物理ページにマッピングします．このときに，複数のページの対応を指定することができます．

マッピングする論理ページと物理ページの対応は，図9-7のように最初の1ワードに論理ページを，次の1ワードに物理ページを指定し，これらのページのペアをそのエントリ数(CXレジスタで指定)分だけ用意して，そのデータ構造へのポインタをDS：SIレジスタで指定します．

ここで，論理ページに対してFFFFHを指定すると，対応する物理ページは解除(アンマップ)されて読み書きができなくなり，そのページを保護することができます．

Map/Unmap Multiple Handle Pages (Logical Page/Segment Address Method) No.17 コード 01H	
機能	複数ページのマップ/アンマップ(論理ページ/セグメント・アドレス方式)
コール	AX＝5001H
	DX←EMMハンドル
	CX←配列内のエントリ数
	DS：SI←配列構造へのポインタ
リターン	AH←ステータス・コード

このファンクションでは，ファンクション1700と同様に，DXレジスタで指定したハンドルの論理ページを指定の物理ページにマッピングします．このときに，複数のページの対応を指定することができ，このファ

〔図9-7〕ファンクション1700におけるページの指定

〔図9-8〕ファンクション1701におけるページの指定

ンクションでは，物理ページをページ番号ではなくセグメント・アドレスで指定します．

マッピングする論理ページとセグメント・アドレス(物理ページ)の対応は，図9-8のように最初の1ワードに論理ページを，次の1ワードにセグメント・アドレスを指定し，これらのページのペアをそのエントリ数(CXレジスタで指定)分だけ用意して，そのデータ構造へのポインタをDS：SIレジスタで指定します．

ここで，論理ページに対してFFFFHを指定すると，対応する物理ページは解除(アンマップ)されて読み書きができなくなり，そのページを保護することができます．

Reallocate Pages	No.18
機能	ページの再割り当て
コール	AH＝51H
	DX←EMMハンドル
	BX←再割り当てのページ数
リターン	AH←ステータス・コード
	BX←再割り当てされたページ数

このファンクションによって，DXレジスタで指定したEMMハンドルに割り当てられている論理ページの数を増やしたり減らしたりすることができます．

再割り当てを要求するページの数は，BXレジスタで指定します．ここで，すでにハンドルに割り当てられているページ数よりもBXレジスタで要求したページ数が大きければ，新たなページが割り当てられます．

このとき，ハンドルに割り当てられている論理ページの順番は，この操作の後も変更されることなく，また新たに割り当てられたページは，以前のページの後に配置(ページ番号がつけられる)されます．

すでにハンドルに割り当てられているページ数よりも，BXレジスタで要求したページ数が小さい場合は，その差分のページ数がページ列の最後から削除されてメモリ・マネージャに返されます．

Get Handle Attribute	No.19 コード 00H
機能	ハンドル属性の取得
コール	AX＝5200H
	DX←EMMハンドル
リターン	AH←ステータス・コード
	AL←ハンドルの属性
	(0：揮発性　1：不揮発性)

このファンクションは，DXレジスタで指定したハンドルに関する属性をALレジスタに返します．

ALレジスタが“0”の場合は，その拡張メモリは揮発性であることを表し，“1”の場合は不揮発性であることを表します．しかし，このファンクションは，メモリ・ボードやシステムのハードウェアに依存し，サポートされていないシステムの場合もあります．

なおPC-9801シリーズでは，揮発性の属性のみが使用可能となっています．

Set Handle Attribute	No.19 コード 01H
機能	ハンドル属性の設定
コール	AX＝5201H
	DX←EMMハンドル
	BL←ハンドルの新しい属性
リターン	AH←ステータス・コード

このファンクションは，DXレジスタで指定したハンドルに関する属性を変更する場合などに使用します．属性の指定はBLレジスタで行い，BLレジスタが“0”の場合は揮発性を指定し，“1”の場合は不揮発性の指定を行います．

なお，PC-9801シリーズでは，不揮発性をサポートしていないため，このファンクションでBLレジスタに“1”を指定するとエラーになります．

Get Attribute Capability	No.19 コード 02H
機能	不揮発性属性のサポート可能性の調査
コール	AX＝5202H
リターン	AH←ステータス・コード
	AL←属性の性能(00H：揮発性のみ)

このファンクションは，メモリ・マネージャが不揮発性の属性をサポート可能かどうかを調べるために使用されます．

結果はALレジスタに返され，もしALレジスタが“0”の場合は，そのシステムは揮発性のみをサポートし，“1”の場合は，揮発性と不揮発性の両方をサポートしているシステムです．

PC-9801シリーズの場合は，不揮発性をサポートしていないので常に“0”が返されます．

Get Handle Name	No.20 コード 00H
機能	ハンドル名の取得
コール	AX＝5300H
	DX←EMMハンドル
	ES：DI←バッファ(8バイト)へのポインタ
リターン	AH←ステータス・コード

このファンクションでは，DXレジスタで指定したEMMハンドルに割り当てられている8文字のハンドル名をES：DIレジスタで指定したバッファに返します．

Set Handle Name	No.20 コード 01H
機能	ハンドル名の設定
コール	AX＝5301H DX←EMM ハンドル DS：SI←ハンドル名へのポインタ
リターン	AH←ステータス・コード

このファンクションでは，DX レジスタで指定されたEMM ハンドルに対して，DS：SI レジスタで指定したハンドル名を割り当てます．ハンドル名は8文字の文字列で指定し，名前に使用する文字に制限はありません．

Get Handle Directory	No.21 コード 00H
機能	ハンドルのディレクトリ情報の取得
コール	AX＝5400H ES：DI←バッファへのポインタ
リターン	AH←ステータス・コード AL←エントリ数(ハンドル数)

このファンクションは，すべてのオープンされているハンドル値とハンドルに割り当てられている名前の情報をES：DI レジスタで指定したバッファに返します．

このとき，バッファに返される情報は，図9-9 のように最初の1ワードにハンドル値が，次の8バイトには，そのハンドルにつけられている名前が返され，これらの情報は，オープンされているハンドルの数だけ続きます．ここで，ハンドル名が割り当てられていないときは，ハンドル名のフィールドはNULL キャラクタで埋められます．

また，バッファのサイズは，一つのハンドルにつき10バイトが必要であり，ハンドルの最大値が00FFH(255)であることから，

〔図9-9〕ファンクション 2100 で返されるディレクトリ情報

10バイト×255＝2550(バイト)

となります．

Search for Named Handle	No.21 コード 01H
機能	指定の名前をもつハンドルの検索
コール	AX＝5401H DS：SI←ハンドル名へのポインタ
リターン	AH←ステータス・コード DX←検索されたEMM ハンドル

このファンクションでは，DS：SI レジスタで指定された文字列をハンドル名として，ハンドル名のディレクトリから検索し，そのハンドル名に対応したEMM ハンドルをDX レジスタに返します．

ここで，DS：SI レジスタで指定するハンドル名は，すべてがNULL キャラクタであってはなりません．

Get Total Handles	No.21 コード 02H
機能	ハンドル総数の取得
コール	AX＝5402H
リターン	AH←ステータス・コード BX←ハンドル総数

このファンクションでは，メモリ・マネージャがサポートしているハンドルの総数(OS 専用の0000H も含む)をBX レジスタに返します．

Alter Page Map & Jump	No.22
機能	ページ・マップの変更とジャンプ
コール	AH＝55H AL←モード (0：ページ番号　1：セグメント) DX←EMM ハンドル DS：SI←ジャンプ・アドレスを含むデータ構造へのポインタ
リターン	AH←ステータス・コード

このファンクションでは，拡張メモリのマッピング状態を変更し，同時に指定のアドレスに制御を渡します．

ここで，DX レジスタにはEMM ハンドルを指定し，DS：SI レジスタには，マッピングとジャンプに必要な情報の入ったデータへのポインタを指定します．DS：SI レジスタの指すデータ構造は，図9-10 に示すように最初のダブル・ワードにはターゲットとなるプログラムへのFAR ポインタを指定します．

このファンクションでは，一度に複数のページをマッピングすることができ，このために，次の1バイトにはマッピングのエントリ数(ページの数)を指定します．

次のダブル・ワードにはマッピング情報の入ってい

るデータ領域の FAR ポインタを指定します．

マッピング情報の構造は，最初の１ワードに論理ページの番号を指定し，次の１ワードに物理ページの番号かセグメント・アドレスを指定します．

ここで，このフィールドのデータが，物理ページの番号なのか，セグメント・アドレスなのかは AL レジスタで指定されるモードに依存します．AL レジスタが“0”の場合はページ番号であり，“1”の場合はセグメント・アドレスを表しています．

これらのマッピング情報は，エントリ数の数だけ続けて指定することができ，これによって，複数のページを一度にマッピングして，その拡張メモリ内の指定されたアドレスに制御が移ります．

ここで，このファンクションでは，目的のアドレスをサブルーチン・コールするのではなく，FAR JUMP することに注意が必要です．

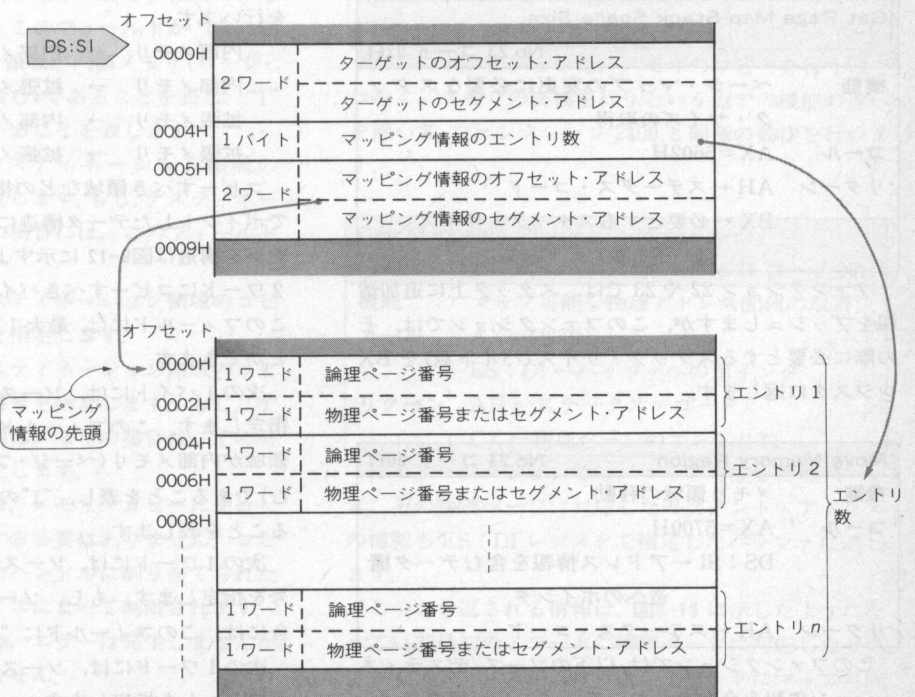
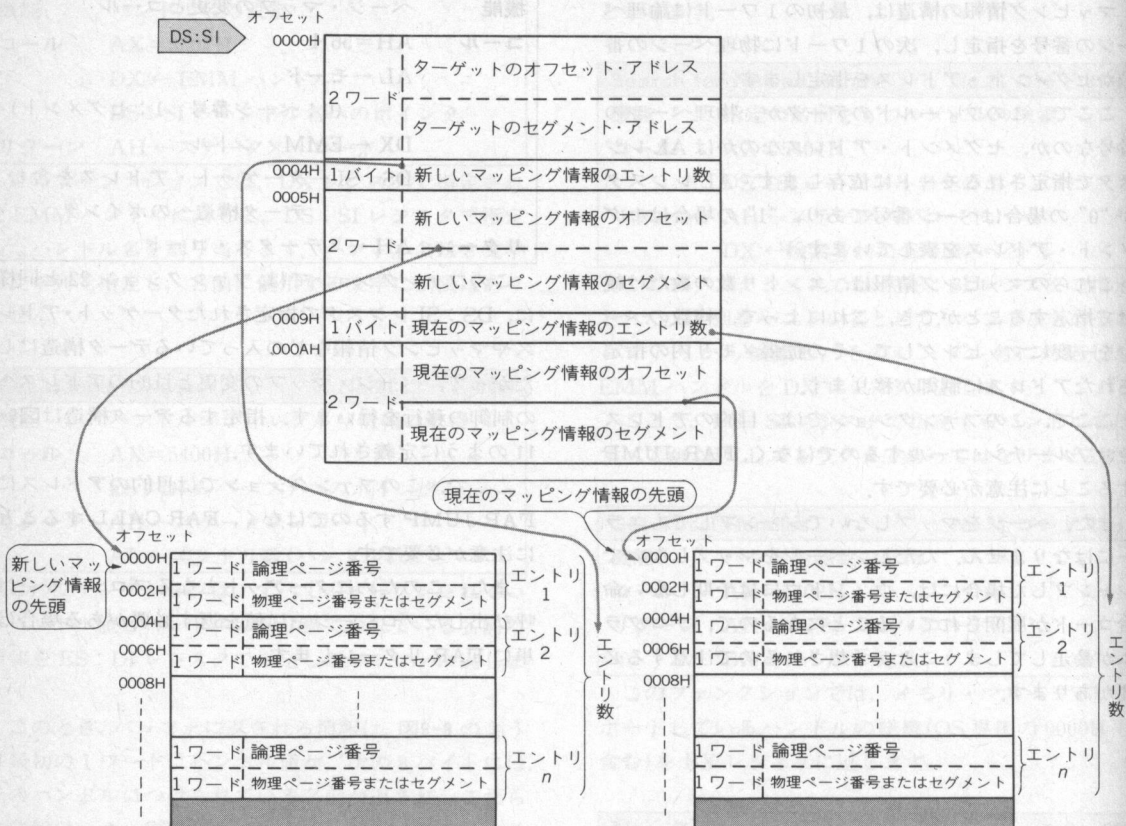
また，ページをマップしないでジャンプしてもエラーにはなりません．ただし，ページをマップしないでジャンプした場合には，ウィンドウには予期しない命令コードが展開されていることになるので，プログラムが暴走してしまうことが予想されるので注意する必要があります．

Alter Page Map & Call		No.23
機能	ページ・マップの変更とコール	
コール	AH＝56H AL ←モード （0：ページ番号　1：セグメント） DX ← EMM ハンドル DS：SI ←ターゲット・アドレスを含むデータ構造へのポインタ	
リターン	AH ←ステータス・コード	

このファンクションでは，ファンクション 22 と同様に，DS：SI レジスタで指定されたターゲット・アドレスやマッピング情報などの入っているデータ構造にしたがって，ページ・マップの変更と目的のアドレスへの制御の移行を行います．指定するデータ構造は図9-11 のように定義されています．

ここで，このファンクションでは目的のアドレスに FAR JUMP するのではなく，FAR CALL することに注意が必要です．

また，このためにターゲットとなるプロシージャは，呼び出したプロシージャに値を返す必要がある場合は，単に FAR リターンします．

〔図9-10〕
ファンクション 22 で指定するデータの構造

〔図9-11〕ファンクション23で指定するデータの構造

Get Page Map Stack Space Size	
	No.23 コード 02H
機能	ページ・マップの変更に必要なスタック・サイズの取得
コール	AX=5602H
リターン	AH←ステータス・コード BX←必要とするスタック・サイズ(バイト数)

ファンクション22や23では，スタック上に追加情報をプッシュしますが，このファンクションでは，その際に必要とするスタック・サイズ(バイト数)をBXレジスタに返します．

Move Memory Region	No.24 コード 00H
機能	メモリ領域の移動
コール	AX=5700H DS：SI←アドレス情報を含むデータ構造へのポインタ
リターン	AH←ステータス・コード

このファンクションでは，以下のソース/デスティネーションの組み合わせにおいて，メモリ領域のコピーを行います．

内部メモリ → 内部メモリ
内部メモリ → 拡張メモリ
拡張メモリ → 内部メモリ
拡張メモリ → 拡張メモリ

コピーすべき領域などの指定は，DS：SIレジスタでポイントしたデータ構造によって指定します．そのデータ構造は図9-12に示すようになっていて，最初の2ワードにコピーすべきバイト数を指定します．なお，このフィールドには，最大1Mバイトまで指定することができます．

次の1バイトには，ソース領域のメモリのタイプを指定します．このフィールドが“0”の場合は，ソース領域が内部メモリ(ページ・フレーム・セグメントを含む)であることを表し，“1”の場合には拡張メモリであることを表します．

次の1ワードには，ソース領域のEMMハンドル番号を指定します．もし，ソース領域が内部メモリの場合には，このフィールドに“0”を設定します．

次の1ワードには，ソース領域のコピーを開始するオフセットを指定します．

〔図9-12〕ファンクション24で使用するソース領域とデスティネーション領域の指定方法

*1：0＝内部メモリ(ページ・フレーム含む)，1＝拡張メモリ
*2：0＝内部メモリ，1＝拡張メモリ

その次の1ワードには，ソース領域のコピーを開始する論理ページ番号を指定します．もし，ソース領域が内部メモリの場合は，そのセグメント・アドレスを指定します．

次の1バイトには，デスティネーション領域のメモリのタイプを指定します．このフィールドが"0"の場合は，デスティネーション領域が内部メモリ(ページ・フレーム・セグメントを含む)であることを表し，"1"の場合には拡張メモリであることを表します．

次の1ワードには，デスティネーション領域のEMMハンドル番号を指定します．もし，デスティネーション領域が内部メモリの場合には，このフィールドに"0"を設定します．

次の1ワードには，デスティネーション領域のコピーを開始するオフセットを指定します．

次の1ワードには，デスティネーション領域のコピーを開始する論理ページ番号を指定します．もし，デスティネーション領域が内部メモリの場合は，そのセグメント・アドレスを指定します．

このファンクションでは，メモリ・コピーに先行してマッピング状態を保存する必要はありません．コピーするバイト数は，指定のハンドルに割り当てられた拡張メモリ・ページのサイズによって制限されます．また，バイト数が0の場合，エラーは発生しませんが，メモリのコピーも行われません．

バイト数が16Kバイトを越えた場合や，複数の論理ページにまたがっている場合にはエラーにはなりませんが，十分な大きさの論理ページが残っている必要があります．

もし，ソース領域とデスティネーション領域が重複している場合，移動の方向が正しく選択されて完全なコピーが行われますが，領域の重複が発生したことを表すステータスが返されます．

〔図9-13〕ファンション2500で返される配列の構造

Exchange Memory Region	No.24 コード 01H
機能	メモリ領域の交換
コール	AX＝5701H
	DS：SI←アドレス情報を含むデータ構造へのポインタ
リターン	AH←ステータス・コード

このファンクションは，メモリのコピーを行うのではなく，メモリの交換を行うという点での機能の違いを除いて，ファンクション2400と同様の動作を行います．

Get Mappable Physical Address Array	No.25 コード 00H
機能	マップ可能な物理アドレス配列の取得
コール	AX＝5800H
	ES：DI←バッファへのポインタ
リターン	AH←ステータス・コード
	CX←物理ページのエントリ数

このファンクションでは，マップ可能な物理ページと，その物理ページに対応したセグメント・アドレスの情報をES：DIレジスタで指定したバッファに返します．

このとき返される情報は，**図9-13**に示したようになっていて最初の1ワードに物理ページに対応したセグメント・アドレスが入り，次の1ワードにマップ可能な物理ページのページ番号が入ります．これらの情報

はCXレジスタに返されたエントリの数だけ続けられます．このバッファのサイズは，次に述べるファンクション2501を用いてマップ可能な物理ページの数を取得することによって計算することができます．

Get Mappable Physical Address Array Entries No.25 コード 01H	
機能	マップ可能な物理アドレス配列のエントリ数の取得
コール	AX＝5801H
リターン	AH←ステータス・コード CX←物理ページのエントリ数

このファンクションでは，ファンクション2500で物理ページに関する情報を格納するためのバッファ・サイズを知りたい際に使用され，CXレジスタにマップ可能な物理ページ数を返します．

Get Hardware Configuration Array No.26 コード 00H	
機能	ハードウェア構成に関する情報の取得
コール	AX＝5900H ES：DI←バッファへのポインタ
リターン	AH←ステータス・コード

このファンクションでは，ES：DIレジスタで指定したバッファに対し，OS/E環境によって使用される拡張メモリのハードウェア構成などの情報を返します．

ハードウェア構成情報は，**図9-14**に示すようになっていて，最初の1ワードにマップ可能のロウ物理ページのサイズをパラグラフ(16バイト)単位で返します．

EMSで使用する標準のページ・サイズは16Kバイトですが，一部のメモリ・ボードでは，この標準サイズよりも小さいサイズをサポートしている場合もあり，

〔図9-14〕ファンクション2600によって得られるハードウェア構成情報

オフセット		
ES:DI → 0000H	1ワード	ロウ物理ページのサイズ(パラグラフ：16バイト単位)
0002H	1ワード	代替マッピング・レジスタの数
0004H	1ワード	マッピング情報の格納に必要なバイト数
0006H	1ワード	DMAチャネルに割り当てられるレジスタ・セットの数
0008H	1ワード	DMAレジスタ・セットの利用モード*
000AH		

*：0＝代替マップ可能，1＝DMAレジスタは1個

その標準よりも小さいサイズをロウ・ページと呼んでいます．したがって，このフィールドはハードウェアの動作レベルからみたマップ可能なページ・サイズが設定されます．なおPC-9801シリーズの場合は，ロウ・ページのサイズも16Kバイトとなっています．

次の1ワードには，代替マッピング・レジスタの数が入ります．すべての拡張メモリ・ボードは，論理ページを物理ページに対応させるために，少なくても一つのハードウェア・レジスタ(マッピング・レジスタ)をもっていますが，拡張メモリ・ボードの中には，一つ以上のマッピング・レジスタをもっているものもあり，その追加されたマッピング・レジスタを代替マッピング・レジスタと呼びます．このフィールドには，この代替マッピング・レジスタの数が返されます．

次の1ワードには，マッピング情報を格納するために必要とするメモリ領域のサイズ(バイト数)が入ります．このフィールドの値は，ファンクション1503(ページ・マップを格納する配列サイズの取得)で返される値と同じ値が返されます．

次の1ワードには，DMAチャネルに割り当てられているレジスタ・セットの数が入ります．

次の1ワードには，DMAレジスタ・セットの利用に関するコードが入ります．このフィールドが"0"であれば，DMAレジスタ・セットはファンクション28と同様に機能します．また，このフィールドが"1"の場合，拡張メモリ・ハードウェアはDMAレジスタ・セットを一つしか所有していません．標準のEMSボードの場合，このフィールドには"0"が設定されます．

Get Unallocated Raw Page Count No.26 コード 01H	
機能	未アロケートのロウ・ページ数の取得
コール	AX＝5901H
リターン	AH←ステータス・コード BX←未アロケートのロウ・ページ数 DX←ロウ・ページの総数

このファンクションは，拡張メモリ内のマップ可能なページ総数(標準サイズでない)と，割り当てられていないページ数(標準サイズでない)を返します．

ある種類の拡張メモリ・ボードは，標準ページ(16Kバイト)の約数となるようなページ・サイズをもつものがあり，その標準ページでない拡張メモリ・ページをロウ・ページと呼んでいます．このファンクションでは，ファンクション03(未アロケート・ページ数の取得)と異なり，ページの単位としてロウ・ページを用いています．

なお，PC-9801シリーズの場合は，ロウ・ページは標準ページと同じサイズ(16Kバイト)となっています．

Allocates Standard Pages	No.27 コード 00H
機能	標準サイズのページの割り当て
コール	AX＝5A00H BX←要求する標準ページ数
リターン	AH←ステータス・コード DX←EMMハンドル

このファンクションでは，OSが要求する数だけの標準サイズのページを割り当てます．このファンクションは，ファンクション04（ページの割り当て）とは異なり，ハンドルにゼロ・ページを割り当てることが可能です．

Allocates Raw Pages	No.27 コード 01H
機能	ロウ・ページの割り当て
コール	AX＝5A01H BX←要求するロウ・ページ数
リターン	AH←ステータス・コード DX←EMMハンドル

このファンクションは，OSが要求する数だけロウ・ページ（標準サイズではない）を割り当てます．このファンクションは，ファンクション04（ページの割り当て）とは異なり，ハンドルにゼロ・ページを割り当てることができます．

Get Alternate Map Register Set	No.28 コード 00H
機能	代替マップ・レジスタ・セットの取得
コール	AX＝5B00H
リターン	AH←ステータス・コード BL←マップ・レジスタ・セットの番号 ES：DI←格納領域へのポインタ

このファンクションは，その時点でアクティブになっているマップ・レジスタによって，以下のいずれかを実行します．

◆ ファンクション2801（代替マップ・レジスタ・セットの設定）がBLレジスタ（代替マップ・レジスタ・セット番号）に“0”を返していた場合

(a) ファンクション2801によって保存されていたマッピング情報へのポインタをES：DIレジスタに返す．

(b) 同時に，そのポインタがゼロでなければ，このファンクションは，現在のマッピング情報をES：DIレジスタの指す格納領域にコピーする．ES：DIレジスタに返されたポインタがゼロであれば，マッピング情報のコピーは行われない．

(c) ここで，ES：DIレジスタの指すポインタは，先行するファンクション2801の呼び出しによって初期設定されていなければならない．また，そのポインタの指す格納領域は，ファンクション1500（ページ・マップの取得）によって初期設定されていなければならない．

◆ ファンクション2801（代替マップ・レジスタ・セットの設定）がBLレジスタに“0”でない代替マップ・レジスタ・セット番号を返していた場合

このファンクションがコールされた時点で使用されている代替マップ・レジスタ・セットの番号がBLレジスタに返される．

Set Alternate Map Register Set	No.28 コード 01H
機能	代替マップ・レジスタ・セットの設定
コール	AX＝5B01H BL←新しいマップ・レジスタ・セットの番号 ES：DI←格納している領域へのポインタ
リターン	AH←ステータス・コード

このファンクションでは，BLレジスタで指定されるマップ・レジスタ・セットの番号にしたがって，次の処理のいずれかが実行されます．

◆ マップ・レジスタ・セット番号がゼロの場合

(a) その代替マップ・レジスタ・セットをアクティブにする．

(b) ES：DIレジスタの内容（マッピング情報格納領域へのポインタ）がゼロでなければ，そのマッピング情報が拡張メモリ・ボード内のマッピング・レジスタにコピーされる．もし，ポインタがゼロであれば内容のコピーは行われない．

◆ マップ・レジスタ・セット番号がゼロでない場合

指定のマップ・レジスタ・セットがアクティブになる．

Get Alternate Map Save Array Size	No.28 コード 02H
機能	代替マップ・セーブ配列のサイズ取得
コール	AX＝5B02H
リターン	AH←ステータス・コード DX←配列のサイズ（バイト数）

このファンクションでは，ファンクション2801および2802で使用されるマッピング情報格納領域のサイズをDXレジスタに返します．

Allocate Alternate Map Register Set	No.28 コード 03H
機能	代替マップ・レジスタ・セットの割り当て
コール	AX＝5B03H
リターン	AH←ステータス・コード BL←マップ・レジスタ・セットの番号

このファンクションでは，ファンクション2801および2802で使用できる代替マップ・レジスタ・セットの番号をBLレジスタに返します．

もし，ハードウェアが代替マップ・レジスタをサポートしていなければ，BLレジスタには“0”が返されます．

Deallocate Alternate Map Register Set No.28 コード 04H	
機能	代替マップ・レジスタ・セットの解放
コール	AX=5B04H BL←マップ・レジスタ・セットの番号
リターン	AH←ステータス・コード

このファンクションでは，BLレジスタで指定された代替マップ・レジスタ・セットをEMMに返します．EMMは必要なときに，この代替マップ・レジスタを再割り当てすることができます．

このファンクションでは，指定の代替マップ・レジスタのマッピング情報を解除することによって，そのページをアクセス不可にして保護する目的でも使用されます．

Allocate DMA Register Set No.28 コード 05H	
機能	DMAレジスタ・セットの割り当て
コール	AX=5B05H
リターン	AH←ステータス・コード BL←DMAレジスタ・セットの番号

このファンクションでは，もしDMAレジスタ・セットが現在使用可能ならば，DMAレジスタ・セットの番号をBLレジスタに返します．

もし，ハードウェアがDMAレジスタ・セットをサポートしていなければBLレジスタには“0”が返されます．

Enable DMA on Alternate Map Register Set No.28 コード 06H	
機能	代替マップ・レジスタによるDMAの使用許可
コール	AX=5B06H BL←DMAレジスタ・セットの番号 DL←DMAチャネル番号
リターン	AH←ステータス・コード

このファンクションでは，BLレジスタで指定された代替マップ・レジスタ・セットを通じて，DLレジスタで指定されたDMAチャネルでのDMAアクセスを可能にします．もし，DMAチャネルかDMAレジスタ・セットにマップされていなければ，そのチャネルに対するDMAは現在のレジスタ・セットを通じてマップされます．

Disable DMA on Alternate Map Register Set No.28 コード 07H	
機能	代替マップ・レジスタに対応するDMAの使用禁止
コール	AX=5B07H BL←マップ・レジスタ・セット番号
リターン	AH←ステータス・コード

このファンクションでは，BLレジスタで指定された代替マップ・レジスタ・セットに対応するDMAチャネルへのDMAアクセスを不可能にします．

Deallocate DMA Register Set No.28 コード 08H	
機能	DMAレジスタ・セットの解放
コール	AX=5B08H
リターン	AH←ステータス・コード BL←DMAレジスタ・セットの番号

このファンクションでは，指定のDMAレジスタ・セットを解放します．このとき，DMAオペレーションで使用できなくなるDMAレジスタ・セットの番号がBLレジスタに返されます．

Prepare Expanded Memory Hardware for Warm Boot No.29	
機能	ウォーム・ブートのための拡張メモリ・ハードウェア準備
コール	AH=5CH
リターン	AH←ステータス・コード

このファンクションでは，緊急のウォーム・ブートのために拡張メモリ・ハードウェアを準備します．これによって，拡張メモリのハードウェアは初期設定されます．

Enable OS/E Function Set No.30 コード 00H	
機能	OS/Eファンクション・セットの使用許可の設定
コール	AX=5D00H BX, CX←アクセス・キー
リターン	AH←ステータス・コード BX, CX←アクセス・キー

このファンクションは，OS/E指定のファンクション(ファンクション26, 28, 30)をすべてのプログラムが使用できるように許可を与えます．この機能は，プログラムがマップ可能な内部メモリ領域に影響を与えるようなファンクションを使用することは許可されていません．

OS/Eがこのファンクションを使用禁止にしている

とき，プログラムがこのファンクションを使用しようとするとエラー・ステータスが返されます．

Disable OS/E Function Set	No.30 コード 01H
機能	OS/Eファンクション・セットの使用禁止の設定
コール	AX＝5D01H BX，CX←アクセス・キー
リターン	AH←ステータス・コード BX，CX←アクセス・キー

このファンクションは，OS/E指定のファンクション(ファンクション26，28，30)をOS/E以外のプログラムが使用できないようにアクセスを禁止します．

Return Access Key	No.30 コード 02H
機能	アクセス・キーのリターン
コール	AX＝5D02H BX，CX←アクセス・キー
リターン	AH←ステータス・コード

このファンクションでは，OS/EがEMMにアクセス・キーを返せるようにする機能をもちます．このBX，CXレジスタで指定するアクセス・キーは，ファンクション3000および3001の実行の際に使用するアクセス・キーと同一のキーとなります．

EMMにアクセス・キーを返すと，EMMはインストール時の状態になります．すなわち，OS/Eファンクション・セットへのアクセスが使用許可になります．

Get Page Frame Status	No.31 コード 00H
機能	ページ・フレーム用バンクのステータス取得
コール	AX＝7000H
リターン	AH←ステータス・コード AL←ステータス (0：ページ・フレームに使用可 1：ページ・フレームに使用不可)

このファンクションは，ページ・フレーム用のバンクの使用状態を返します．ALレジスタに返されたステータスが“0”の場合には，そのバンクが拡張メモリのページ・フレームとして使用可能なことを表し，“1”の場合にはページ・フレームとして使用できません．

Enable/Disable Page Frame	No.31 コード 01H
機能	ページ・フレーム用バンクの状態の設定
コール	AX＝7001H BL←指示 (0：ページ・フレーム　1：VRAM)
リターン	AH←ステータス・コード AL←ステータス (0：ページ・フレームに使用可 1：ページ・フレームに使用不可)

このファンクションは，指定された状態にページ・フレーム用のバンクの切り替えを行います．

＊　　　＊

ここでは，MS-DOS ver.3.30で機能追加された拡張メモリについて解説しました．MS-DOSの拡張メモリは，まだその仕様が発表されたばかりであり，その評価やアプリケーション・プログラムの開発には，いましばらくの時間がかかりそうです．

本章でも述べたように，筆者の憶測では，この拡張メモリにはMS-DOSの将来がかかっているといっても過言ではないかもしれません．なぜなら，MS-DOSの640Kバイトの壁はまさにネックであり，一方ではMS-DOSのマルチタスク化への期待も強く，どうしても大きなメモリ空間が必要となるからです．

現在の拡張メモリは，OSが関与していないので単なるデバイスとして動作しています．したがって，拡張メモリへのアクセスも比較的容易にテストすることができます．

我々MS-DOSユーザは，これから発展するであろう拡張メモリの仕様を，実際にアクセスしながら理解しておくことが，将来的にも重要なことといえるでしょう．

Appendix A／PC-9801シリーズのBIOS

ここでは，キャラクタ型デバイス・ドライバの例であるグラフィック・コンソール・ドライバ(Graphic Console Driver：以下GCドライバ)を作成するための基礎として，PC-9801のBIOSについて整理します．

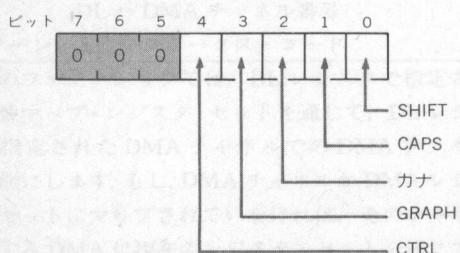
PC-9801シリーズでは，グラフィックスの制御ルーチンがBIOS(Basic I/O System)に組み込まれているのでこれを利用します．また，MS-DOSは，キーボードのタイプアヘッド・バッファに文字が入っているかどうかを確認するために，ドライバに対してステータス・チェックのコマンドを発行します．この際に，キーボード・ステータスを調べるのにやはりPC-9801シリーズのBIOSを利用します．

PC-9801シリーズでは，CRTやキーボードおよびマウスなど，各種の周辺機器に対して専用の制御プログラムがROMにBIOSとして組み込まれており，ユーザが自由に利用することができます．

● キーボードBIOS

キーボードBIOSは，キー・バッファの状態を調べたり，キー・データの読み出しを行ったりします．キーボードBIOSは，ソフトウェア割り込みのINT 18Hによって呼び出すことができます．

Read Key Data	00H
機能	キー・データの読み出し
コール	AH＝00H
リターン	AH←スキャン・コード AL←内部コード

このファンクションでは，キー・データ・バッファの先頭に格納されているキー・データのコード(特殊なコード)を読み出します．このファンクションでは，キー・データ・バッファ内にキー・データが格納されていなければキー入力されるまで待ちます．

得られたキー・データは，JISコードやASCIIコードのような一般的なコードではなく，PC-9801シリーズのキーボードに割り付けられた特殊なコードです．

Read Status	01H
機能	キー・データ・バッファの状態の読み出し
コール	AH＝01H
リターン	AH←スキャン・コード AL←内部コード BH←ステータス 00H：データなし 01H：データあり

このファンクションでは，キー・データ・バッファ内に格納されているキー・コードを調べます．もし，バッファ内にデータがなければBHレジスタに00Hを返します．このとき，AXレジスタに返されたキー・コードは無効となります．

もし，BHレジスタに返された値が01Hの場合は，AXレジスタに返された内容は，キー・データ・バッファの内容であり有効です．このファンクションでも，AXレジスタに返されるキー・データはPC-9801の特殊なコードです．

また，このファンクションが実行されてもバッファのポインタは変更されません．

Read Shift Key	02H
機能	シフト・キー状態の読み出し
コール	AH＝02H
リターン	AL←シフト・キーの状態

このファンクションは，現在押されているシフト・キーの状態を調べるためにコールされます．ここで，ALレジスタに返されるデータの各ビットの意味は図A-1のようになっています．

〔図A-1〕ALレジスタの内容

Initialaize Key Board	03H
機能	キーボードI/Fの初期化
コール	AH＝03H
リターン	なし

このファンクションは，キーボードBIOSが使用しているワーク・エリアと，ハードウェアの初期化を行います．

Read Key Group Status	04H
機能	キー・コード・グループの状態の読み出し
コール	AH＝04H AL ←キー・コードのグループ番号
リターン	AH ←キー・インの状態

このファンクションでは，AL レジスタで指定されたキー・コード・グループの入力状態を調べ，その状態を AH レジスタに返します．ここで使用されるグループ番号やキー・コードに関しては割愛します．

● グラフ LIO

PC-9801 シリーズの BIOS では，より論理的なグラフィックス処理ルーチンが ROM 内に収められており，この処理ルーチンをグラフ LIO と呼んでいます．グラフ LIO には，**表A-1** のように 17 種のコマンドが用意されていて，通常，割り込み番号 A0H～AFH および CEH の割り込みベクタを介してコールされます．

また，これらの割り込みエントリは**図A-2** のように ROM 上の F9906H 番地から 4 バイトおきに格納されています．したがって，このグラフ LIO のユーザは，あらかじめこれら各コマンドの割り込みエントリを，各割り込みベクタ・テーブルにセットしておかなければなりません．

今回使用した MS-DOS システムでは，割り込み番号 A0H～AFH は使用されていないので，グラフ LIO の割り込みベクタをセットするだけにしましたが，すでに A0H～AFH の割り込みベクタが使用されているような場合には，一度ユーザのバッファにこれらのベクタ・テーブルを格納しておき，グラフィックス処理が終わった時点で元のベクタ・テーブルに復元する必要があります．

また，グラフ LIO のユーザは，これら A0H～AFH，CEH 以外にも注意しなければならない割り込み番号があります．

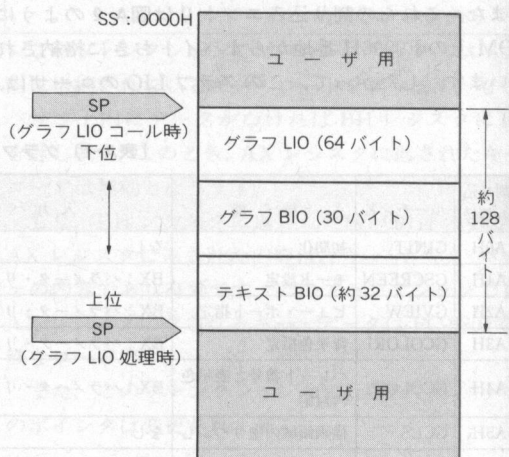
グラフ LIO では比較的時間のかかるコマンド実行中にも，ほかの割り込み処理が行えるように，時々割り込み番号 C5H のルーチンをコールしています．これによって，ユーザがこの C5H 割り込みルーチンの中で，たとえば STOP キーの監視やほかの割り込み処理を行うことなども可能になっています．

〔表A-1〕グラフ LIO の入出力パラメータ

割り込み番号	ルーチン名	機　能	入力パラメータ	出力パラメータ
A0H	GINIT	初期化	なし	AH＝00H：正常終了
A1H	GSCREEN	モード設定	BX：パラメータ・リストへのポインタ	AH＝00H：正常終了，AH＝05H：不正呼び出し
A2H	GVIEW	ビュー・ポート指定	BX：パラメータ・リストへのポインタ	AH＝00H：正常終了，AH＝05H：不正呼び出し
A3H	GCOLOR1	背景色指定	BX：パラメータ・リストへのポインタ	AH＝00H：正常終了
A4H	GCOLOR2	パレット番号と表示色の対応	BX：パラメータ・リストへのポインタ	AH＝00H：正常終了
A5H	GCLS	描画領域の塗りつぶし	なし	AH＝00H：正常終了
A6H	GPSET	点を打つ	BX：パラメータ・リストへのポインタ AH＝01H：フォアグラウンド・パレット番号 AH＝02H：バックグラウンド・パレット番号	AH＝00H：正常終了
A7H	GLINE	直線，矩形を描く	BX：パラメータ・リストへのポインタ	AH＝00H：正常終了
A8H	GCIRCLE	円，楕円を描く	BX：パラメータ・リストへのポインタ	AH＝00H：正常終了，AH＝06H：演算オーバフロー
A9H	GPAINT1	色で塗りつぶし	BX：パラメータ・リストへのポインタ	AH＝00H：正常終了，AH＝05H：不正呼び出し， AH＝07H：ワーク・エリア不足
AAH	GPAINT2	タイルで塗りつぶし	BX：パラメータ・リストへのポインタ	AH＝00H：正常終了，AH＝05H：不正呼び出し， AH＝07H：ワーク・エリア不足
ABH	GGET	描画情報の格納	BX：パラメータ・リストへのポインタ	AH＝00H：正常終了，AH＝05H：不正呼び出し
ACH	GPUT1	描画情報の表示	BX：パラメータ・リストへのポインタ	AH＝00H：正常終了，AH＝05H：不正呼び出し
ADH	GPUT2	日本語の描画	BX：パラメータ・リストへのポインタ	AH＝00H：正常終了，AH＝05H：不正呼び出し
AEH	GROLL	描画画面の移動	BX：パラメータ・リストへのポインタ	
AFH	GPOINT2	ドットのパレット番号の取得	BX：パラメータ・リストへのポインタ	AH＝00H：正常終了，AL：パレット番号
CEH	GCOPY	ドット情報の格納	AX：左上点 X 座標 BX：左上点 Y 座標 CL：X 方向ドット数 CH：Y 方向ドット数 DI：バッファのオフセット・アドレス ES：バッファのセグメント・アドレス	AH：不定

〔図A-2〕割り込みエントリ・テーブル

〔図A-3〕グラフ LIO ワーク・エリア

〔図A-4〕グラフ LIO スタック・エリア

次に，グラフ LIO を使用する場合の準備として，割り込みベクタ・テーブルのセットに加えて，ワーク・エリアの確保もしなければなりません．

グラフ LIO のワーク・エリアは図**A-3** のように GCOPY(INT　CEH ルーチン)を使用しない場合に 1200H バイト，GCOPY ルーチンを使用する場合は，1400H バイトを用意しなければなりません．

そして，このワーク・エリアは，必ずデータ・セグメント(DS レジスタ)のオフセット 0000H 番地から配置されなければなりません．このワーク・エリアのアドレスは，GINIT(INT A0H ルーチン)コマンドをコールする際にグラフ LIO に渡され，以後の各コマンドをコールする際にも，DS レジスタはこのワーク・エリアを指していなければなりません．

また，パラメータ・リストは，このデータ・セグメント内に配置されていなければならず，そのポインタとしては，BX レジスタが使われています．

これらのグラフ LIO の各コマンドでは，DS, SS, SP の 3 個のレジスタは保証されていますが，そのほかのレジスタは保存されないので注意が必要です．

また，グラフ LIO のスタック・エリアとして，図**A-4** のように約 100H バイト(同図はスタック内の概念的な使用状況であり，その内容は順不同になる)を用意しなければなりません．

これらのグラフ LIO の各コマンドで使用されるパレット・コードやカラー・コード，そのほかのパラメ

〔図A-5〕
GSCREEN のパラメータ・フォーマット

パラメータ	画面モード	画面スイッチ	アクティブ・ページ	ディスプレイ・ページ
00H	カラー・モード (640×200 ドット)	表示あり	アクティブ・ページの番号 (0〜11)	ディスプレイ・ページの番号 (0〜31)
01H	モノクロ・モード (640×200 ドット)	表示あり 高速書き込み		
02H	モノクロ・モード (640×400 ドット)	表示なし		
03H	カラー・モード (640×400 ドット)	表示なし 高速書き込み		
FFH		今までのモードを引き継ぐ		

〔図A-6〕 GVIEW のパラメータ・フォーマット

〔図A-7〕 GCOLOR1 のパラメータ・フォーマット

BX +0	+1	+2	+3
(未使用)	バックグラウンド・カラー (0〜7)	ボーダー・カラー (0〜7)	フォアグラウンド・カラー (0〜7)

ータや用語については，BASIC コマンドのそれとほぼ互換性があるので，ここでは省略します．必要な場合は BASIC マニュアルなどを参照してください．

また，カラー・コードなどは，対象機種として最もベースとなる PC-9801E/F タイプを想定しているので，パラメータ・リストの説明では 8 色になっていますが，機種によって 16/4096 色モードの場合は多少の変更を要します．そして，座標点などのパラメータは，矛盾のないように指定する必要があります．

◆ 初期化　　(GINIT：A0H)

このルーチンでは，グラフ LIO(ワーク・エリアや GDC など)の初期化が行われます．グラフ LIO を使用する場合は，最初に必ずこのコマンドを実行しなければなりません．

◆ モード設定　　(GSCREEN：A1H)

グラフィック画面のモード設定を行います(図A-5)．

◆ 描画領域の指定　　(GVIEW：A2H)

アクティブ画面内の描画領域(ビューポート)の指定を行います．このコマンドでは，必要に応じてビューポート内の塗りつぶしや外枠の描画も行われます(図A-6)．

また，このコマンド実行後はアクティブ画面への図形描画はビューポート内にのみ反映されることになります．

◆ 背景色などの指定　　(GCOLOR1：A3H)

バックグラウンド・カラー，ボーダ・カラー，フォアグラウンド・カラーの指定を行います(図A-7)．

バックグラウンド・カラーとは，グラフィック画面

の地の色で，このあとの GCLS 命令が実行されたときに，ここで指定された色に変わります．ボーダ・カラーは 640×400 ドット・モードの場合は意味がありません．フォアグラウンド・カラーは，パレット番号省略時に使用される色です．

また，PC-9801V シリーズでは，同図のパラメータの後にさらに 1 バイトのパレット・モードのコードが追加されます．

◆ パレット番号と表示色の対応(GCOLOR2：A4H)

パレット番号を指定された表示色コードに対応させます(図A-8)．

これにより，すでに描画済みの表示色も変更した表示色になります．また，PC-9801V シリーズの 16/4096 モードでは表示色コードは 3 バイトが必要になります．

◆ 描画領域の塗りつぶし　(GCLS：A5H)

アクティブ画面内の描画領域をバックグラウンド・カラーで塗りつぶします．このコマンドにはパラメータはありません．

◆ 点の描画　(GPSET：A6H)

指定された座標に指定された色でドットを描きます(図A-9)．

◆ 直線/矩形の描画　(GLINE：A6H)

指定された 2 点を結ぶ直線，あるいはこれを対角線とする矩形の描画を行います．また，必要によりライン・スタイルや矩形内の塗りつぶし(色やタイル・パターン)も行うことができます(図A-10)．

ライン・スタイルやタイリング・データの詳しいことについては BASIC マニュアルを参照してください．

◆ 円/楕円の描画　(GCIRCLE：A7H)

指定された中心点，半径(X 方向，Y 方向)の円や楕円の描画を行います．

また，開始点，終了点の指定により円弧や扇型も描画でき，これらの円や扇型の内部を指定した色，またはタイルで塗りつぶすこともできます(図A-11)．

◆ ペインティング　(GPAINT1：A9H)

指定した点と境界色で決定される領域を，指定した色で塗りつぶします(図A-12)．

同図でのワーク域の必要な大きさはペインティング領域の大きさにより変化するので，ある程度大きめに確保します．

◆ タイリング　(GPAINT2：AAH)

指定した点と境界色で決定される領域を指定されたタイル・パターンで塗りつぶします(図A-13)．

〔図A-8〕GCOLOR2 のパラメータ・フォーマット

〔図A-9〕GPSET のパラメータ・フォーマット

〔図A-10〕GLINE のパラメータ・フォーマット

BX +0 ~ +1	+2 ~ +3	+4 ~ +5	+6 ~ +7	+8	+9	+10	+11	+12
始点の X 座標 (X_1)	始点の Y 座標 (Y_1)	終点の X 座標 (X_2)	終点の Y 座標 (Y_2)	パレット番号 1 (0~7)	描画指定	ライン・スタイル・スイッチ	パレット番号 2 / ライン・スタイル (下位)	ライン・スタイル (上位)

+13	+14 ~ +15	+16 ~ +17
タイル・パターン長	タイル・パターン格納域 オフセットアドレス	タイル・パターン格納域 セグメントアドレス

描画指定 (+9)
- 0：直線
- 1：矩形
- 2：矩形塗りつぶし

ライン・スタイル・スイッチ (+10)
- 00H：指定なし
- 01H：ライン・スタイルまたはパレット番号 2 指定あり
- 02H：タイル・パターン指定あり

パレット番号 2 (+11)
- 矩形内部の塗りつぶし色指定
- 描画コード＝02H のみ有効

ライン・スタイル (+11, +12)

b_0 … b_7	b_8 … b_{15}
下位 (+11)	上位 (+12)

◆ **描画情報の格納**　　　　　　　　(GGET：ABH)

指定領域の描画情報を指定されたバッファに格納します(図A-14).

◆ **格納された描画情報の表示**　　　(GPUT1：ACH)

格納されている描画情報を指定された領域に描画します(図A-15).

◆ **日本語の描画**　　　　　　　　(GPUT2：ADH)

JIS コードで指定された日本語を描画します(図A-16).

〔図A-11〕GCIRCLE のパラメータ・フォーマット

フラグ (+9)
- b_0：開始点指示 (0：なし，1：あり)
- b_1：開始線分指示 (0：なし，1：あり)
- b_2：終了点指示 (0：なし，1：あり)
- b_3：終了線分指示 (0：なし，1：あり)
- b_4：描画方法 (0：全楕円を描画，1：1点のみ描画)
- b_5：塗りつぶし指示 (0：なし，1：あり)
- b_6：タイル・パターン指示 (0：なし，1：あり)

〔図A-12〕GPAINT1 のパラメータ・フォーマット

＊：ワークエリアは 16 バイト以上
＊：ワークエリアはデータ・セグメント内にあること

〔図A-13〕GPAINT2 のパラメータ・フォーマット

＊：ワークエリアは 16 バイト以上
＊：データ・セグメント内にあること

〔図A-14〕GGET のパラメータ・フォーマット

◆ **描画画面の移動**　　(GROLL：AEH)

アクティブ画面全体の描画情報を指定されたドット数だけ上下左右方向にスクロールします(図A-17).

◆ **ドットのパレット番号の通知**　(GPOINT2：AFH)

指定座標のドットのパレット番号を AL レジスタに返します(図A-18).

〔図A-15〕GPUT1 のパラメータ・フォーマット

〔図A-16〕GPUT2 のパラメータ・フォーマット

〔図A-17〕GROLL のパラメータ・フォーマット

BX +0 ~ +2	+2 ~ +4	+4 ~ +5
上下ドット数 (−399~399)	左右ドット数 (−639~639)	フラグ

フラグ (+4)

- 00H：移動後の残り領域をパレット 0 にする
- 01H：移動後の残り領域をバックグラウンド・カラーにする

〔図A-18〕GPOINT2 のパラメータ・フォーマット

BX +0 ~ +2	+2 ~ +4
X 座標 (X)	Y 座標 (Y)

Appendix B グラフィック・コンソール・ドライバの作成

第8章でも述べたように，キャラクタ型デバイスでは，デバイス名が同一の場合，後で追加されたデバイス・ドライバが優先的にアクセスされます．したがって，もしグラフィック・コンソール・ドライバのデバイス名をCONとして登録すれば，新たにデバイス・オープンすることもなく，MS-DOS標準のコンソール・デバイスと同様に，リダイレクトやパイプなどにおいてもアクセス可能になります．

すなわち，たとえばC言語のfprintf関数ではなく，printf関数やputs関数を用いてグラフィックスの制御が可能になるわけです．このことは，Cに限らず言語が何であれ特別のライブラリなどをリンクする必要もなく，グラフィックス処理の記述が可能になることになります．

グラフィックスに関しては，MS-DOSがサポートしていないため，どうしてもマシンに依存してしまうことになります．ここでは，最もポピュラなPC-9801シリーズのグラフィックスに対応することにします．

グラィックス処理では，前に述べたPC-9801シリーズのグラフLIOを利用します．

● グラフィック・コンソール・ドライバの処理

ここでは，グラフィック・コンソール(GC)・ドライバのソース・プログラムについて解説します．GCドライバは，標準入出力(CON)デバイスとして機能し，文字列を用いてグラフィックスの制御を行います．このため，グラフィックス処理を行う場合は，文字列の先頭にグラフィックス制御文字列のインジケータをつける必要があります．

ここでは，ESCコード(1BH)にベル・コード(07H)がつづいたら，その後の文字列はグラフィック制御文字列を表すことにし，その文字列は改行コードによって終了することにします(図B-1)．

もし，文字列がグラフィックス制御文字列でない場合は，通常のCONデバイスと同様にその文字列をスクリーンに表示します．

コンソール制御に関しては，オリジナルのCONデバイスがすでにOEMメーカによってio.sysの中に組み込まれているのでこれを利用します．また，グラフィックス処理に関しては，すでに述べたグラフLIOを呼び出して処理します．

したがって，GCドライバでは，文字列をコンソールに送るべきかグラフLIOに送るべきかの判断と，パラメータの処理が主な仕事になります．

ここで，GCドライバはアセンブリ言語とC言語によって記述してあり，C言語を用いてキャラクタ型デバイス・ドライバを作成する際の参考になるようにしています．一般に，デバイス・ドライバはアセンブリ言語を用いて記述されます．ドライバをアセンブリ言語を用いて記述すると，ドライバ自体の構造は把握しやすくなりますが，ドライバが複雑になればなるほど，その制御構造が理解しにくくなるという欠点をもっています．

一方，ドライバをC言語で記述すると，逆にその制御構造はわかりやすいものとなりますが，CPUレジスタとの対応やドライバ特有の構造がわかりにくくなります．

このため，ここではドライバ構造に関する部分や，CPUレジスタとのやりとりを行う部分はアセンブリ言語で記述し，制御構造を含むドライバの主要な部分はC言語(MS-C ver.4.0)を用いて記述しています．

C言語を用いてデバイス・ドライバを記述する際に，コンパイラに付属の標準ライブラリ関数は利用できません．なぜなら，標準ライブラリ関数では，Cスタートアップ・ルーチンで設定されるグローバル変数をアクセスしたり，入出力に関して標準入出力(CON)をアクセスしているからです．

また，ここではグラフLIOのパラメータ・チェック(カラー・コードなど)は行っていませんが，グラフLIOの不当なアクセスを防ぐため，これらのパラメータのチェック・ルーチンを組み込んだほうが，より実用的なプログラムになるでしょう．

● グラフィック・コンソール・ドライバの構成

GCドライバは，図B-2に示したように6個のソース・ファイルから構成されています．同図では，ドライバ本体のほかにドライバのデバッグ用ソース・ファイルおよび，ドライバ完成後のアクセス(デモンストレーション)用プログラムの作成方法までを示していま

〔図B-1〕グラフィック制御文字列

1BH	07H	制御文字列 (line や color など)

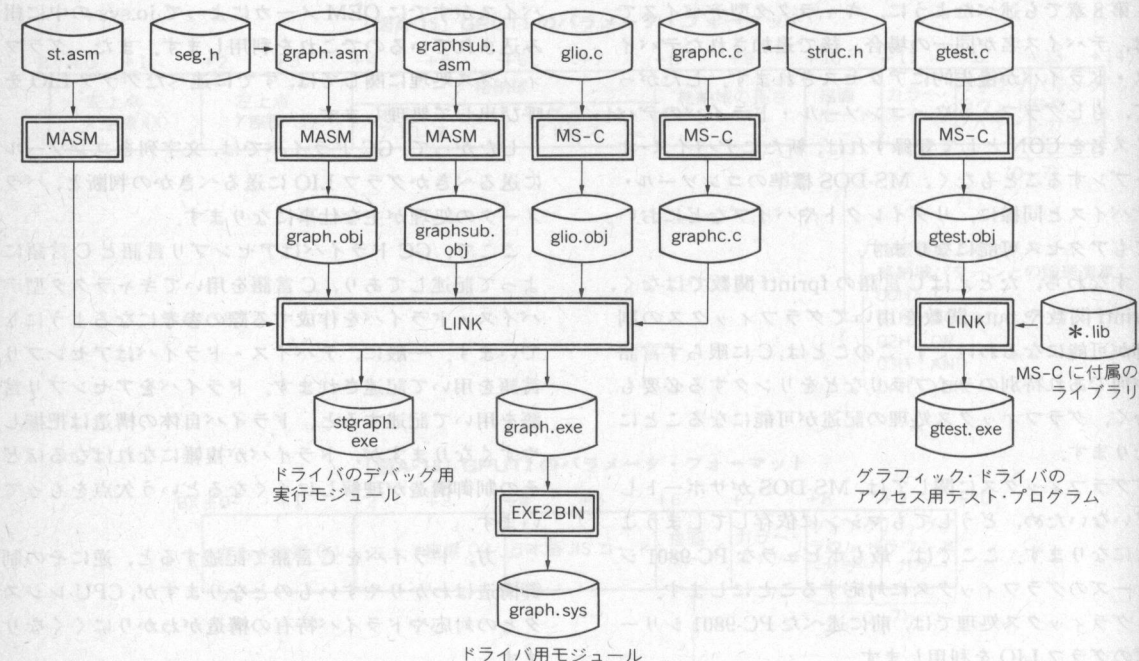
インジケータ (1BH 07H)

す．これらのプログラムの作成は，**リストB-1**のMAKEファイルとMAKEユーティリティによって自動的に行われます．

それぞれのソース・ファイルの機能は以下のとおりです．

① seg.h…セグメント配置の制御を行うためのヘッダ・ファイル．

② graph.asm…GCドライバのメイン・ルーチンとも

〔図B-2〕グラフィック・コンソール・ドライバのモジュール構成

〔リストB-1〕
ドライバ作成用MAKEファイル

```
 1: TRG = graph
 2: OBJ = $(TRG).obj $(TRG)c.obj $(TRG)sub.obj glio.obj
 3: LNK = $(TRG) $(TRG)c $(TRG)sub glio
 4:
 5: st.obj: st.asm
 6:     masm /ZI/Z/ML st,, st;
 7:
 8: $(TRG).obj: $(TRG).asm seg.h
 9:     masm /ZI/Z/ML $(TRG),, $(TRG);
10:
11: $(TRG)sub.obj: $(TRG)sub.asm
12:     masm /ZI/Z/ML $(TRG)sub,, $(TRG)sub;
13:
14: $(TRG)c.obj: $(TRG)c.c struc.h
15:     msc  $(TRG)c.c /Zi /Ze /Zl /Zp /Gs /Fa /DLINT_ARGS /J;
16:
17: glio.obj: glio.c
18:     msc glio.c /Zi /Ze /Zl /Zp /Gs /Fa /DLINT_ARGS /J;
19:
20: gtest.obj: gtest.c
21:     msc gtest.c /Zi /Ze /Zp /Fa /DLINT_ARGS /J;
22:
23: $(TRG).exe: $(OBJ)
24:     link /CO /NOI /MAP /LI $(LNK);
25:
26: st$(TRG).exe: st obj $(OBJ)
27:     link /CO /NOI /MAP /LI st $(LNK), st$(TRG), st$(TRG);
28:
29: gtest.exe: gtest.obj
30:     link /CO /NOI /MAP /LI gtest, ,gtest;
31:
32: $(TRG).sys: $(TRG).exe
33:     exe2bin $(TRG).exe $(TRG).sys
```

〔リストB-2〕
ヘッダ・ファイル seg.h

```
 1: ;*******************************************************************
 2: ;
 3: ;   機    能:  セグメント配置の制御
 4: ;   生    成:  masm /ML seg;
 5: ;
 6: ;*******************************************************************/
 7:             PAGE      60, 130
 8: _TEXT       SEGMENT WORD PUBLIC 'CODE'
 9: _TEXT       ENDS
10: _DATA       SEGMENT WORD PUBLIC 'DATA'
11: _DATA       ENDS
12: CONST       SEGMENT WORD PUBLIC 'CONST'
13: CONST       ENDS
14: _BSS        SEGMENT WORD PUBLIC 'BSS'
15: _BSS        ENDS
16: c_common    SEGMENT WORD PUBLIC 'BSS'
17: c_common    ENDS
18: _dend       SEGMENT WORD PUBLIC 'BSS'
19: _dend       ENDS
20: DGROUP      GROUP     _TEXT, _DATA, CONST, _BSS, c_common, _dend
```

いえるプログラムで，デバイス・ヘッダやストラテジ・ルーチンおよび割り込みエントリ・ルーチンを含む．

③ struc.h…graphc.c ファイルで取り込まれるヘッダ・ファイルで，デバイス・ヘッダやリクエスト・ヘッダおよびコマンド・パケットなどのデータ構造が定義されている．

④ graphc.c…デバイス・コマンド・コードの解析と各コマンド・コードに対応した処理を行う．

⑤ glio.c…グラフ LIO のアクセスを行う．

⑥ graphsub.asm…アセンブリ言語サブルーチンで，主としてシステム・コールや CPU レジスタ値の読み出し/設定を行う．

⑦ st.asm…GC ドライバをデバッグする際のスタートアップ・ルーチンとなり，コマンド・パケットを作成してデバイス・ドライバをアクセスするために，MS-DOS をエミュレーションする．

⑧ gtest.c…GC ドライバの完成後にそのアクセスを行うデモ用プログラム．

以下，これらのファイルごとに，その機能と動作について解説していきます．

● seg.h

リストB-2 のヘッダ・ファイルは，セグメント配置の制御を行います．GC ドライバは，アセンブリ言語と C 言語で記述されていますが，デバイス・ドライバは 64 K バイト以下でなければならず，またセグメントを参照するコードを含んでいてはなりません．

MS-C では，スモール・モデルの場合でも，コード・セグメントとデータ・セグメントは分けて扱っているため，実行時のデータ・セグメント(DS レジスタ値)を計算するのが少々面倒になります．

このため，GC ドライバではディレクティブ GROUP を用いてすべてのセグメントをグループ化し，スタック・セグメントも合わせて 64 K バイトの中に収めています．これによって，プログラム中で特にセグメント・レジスタを意識する必要がなく，8 ビット・マシンにおけるプログラムと同様にリニア・アドレスとしてプログラミングできることになります．

また，セグメント _dend は，GC ドライバの終了アドレスを知るために定義されているセグメントで，リンクされたときにそのセグメントが最後に配置されるようにセグメント名を決めています(**リストB-3 参照**)．

● 8086 vs 68000(その 6)　セグメントの副作用 ●

MS-DOS では，8086 のセグメントの副作用で，C などの高級言語においても CPU のアーキテクチャが見えかくれします．

たとえば，コンパイル・オプションにメモリ・モデルが存在します．また，ポインタでも far ポインタと near ポインタを区別してプログラミングしなければなりません．本来，高級言語は CPU のアーキテクチャを意識しないでプログラミングできるから「高級」言語なのです．

一方，68000 の場合は，メモリ・モデルなどは存在しません．そして，汎用レジスタが 32 ビット長であるため，int と long も区別しなくてよいのです．配列の大きさにも制限がありません．

まさに「高級」言語ではありませんか．

〔リストB-3〕MAP ファイル

```
LINK : warning L4021: no stack segment

 Start  Stop   Length Name                          Class
 00000H 02801H 02802H _TEXT                         CODE
 02802H 0290BH 0010AH _DATA                         DATA
 0290CH 0290CH 00000H CONST                         CONST
 0290CH 0290CH 00000H _BSS                          BSS
 0290CH 02B0DH 00202H c_common                      BSS
 02B0EH 02B0EH 00000H _dend                         BSS    ← 最後に配置されることを確認する

 Origin   Group
 0000:0   DGROUP

  Address          Publics by Name

 0000:227E         _atoi
 0000:28A4         _Bgn_flag
      .
      .
 0000:1F59         _write_status
 0000:9876  Abs    __acrtused

  Address          Publics by Value

 0000:0000         _dev_header
 0000:0012         _glio_buff
      .
      .
 0000:2B0C         Str_ptr
 0000:2B0E         _d_end  ← このラベルが最後に配置されることを確認する
 0000:9876  Abs    __acrtused

Line numbers for GRAPH.OBJ(graph.ASM) segment _TEXT
```

● graph.asm

リストB-4 の graph.asm モジュールは，GC ドライバのメイン・モジュールであり，デバイス・ヘッダやストラテジ・ルーチンおよび割り込みエントリ・ルーチンが記述されています．

デバイス・ドライバの最終アドレスを示すラベル d_end は，セグメント _dend 内で定義され，graphc.c モジュールから参照可能とするため PUBLIC 宣言を行っておきます．また，ラベル entry や strategy も，デバッグ用スタートアップ・ルーチン(st.asm)をリンクした際に参照可能とするため，PUBLIC 宣言を行っておきます．

コード・セグメントでは，まずデバイス・ヘッダの定義を行います．デバイス・ヘッダの先頭のダブル・ワードはデバイス・リンク用のポインタであり，通常，このフィールドには FFFFH を設定しておきます．

次のワード値は，デバイス属性を示すフィールドです．GC ドライバはキャラクタ・デバイスであり，また標準入出力デバイスでもあるので，このフィールドには 8003H を設定しています．

次の二つのワード値は，それぞれストラテジ・ルーチンへのポインタと割り込みエントリ・ルーチンへのポインタが入ります．

そして，デバイス・ヘッダのデバイス名フィールドには CON を設定して，標準入出力デバイスとして使用可能にしています．

デバイス・ヘッダの定義が終わったら，次にグラフ LIO 用のワーク・エリアの確保を行います．すでに述べたように，グラフ LIO のワーク・エリアは，ドット・パターンの格納(INT CEH)を行う場合で 1400H バイト，これが必要ない場合でも 1200H バイトを必要とします．

またマニュアルによると，このワーク・エリアの先頭バイトは未使用領域ということになっていますが，実際にアクセスして確認すると先頭の数バイトは使用されているようです．したがって，ワーク・エリアの先頭(すなわちデバイス・ドライバの先頭)を，DS レジスタに設定してグラフ LIO をアクセスすると，デバイス・ヘッダの部分がオーバライトされて破壊され，システムがハングアップしてしまうことになります．

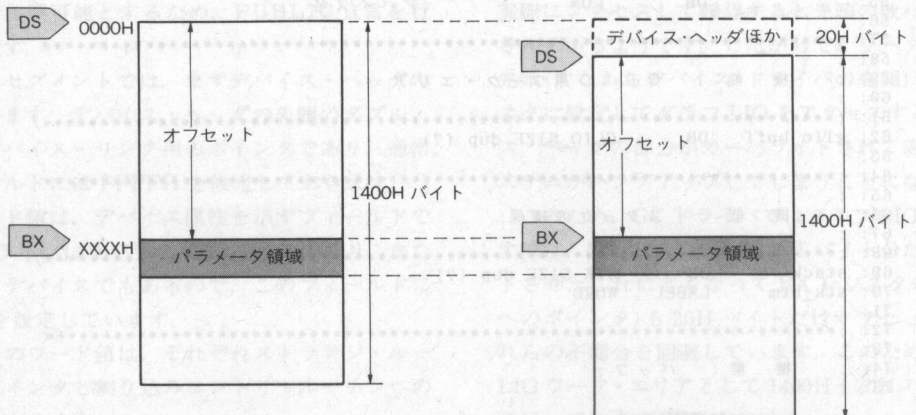
このため，GC ドライバではグラフ LIO に制御を移す際に，DS レジスタ値を 20H バイト分だけオフセットさせ，これにともなって BX レジスタ(パラメータへのポインタ)も 20H バイトだけオフセットさせてこれらの不都合を回避しています．このために，グラフ LIO ワーク・エリアとして 1400H＋20H バイトの確保を行っています(図B-3)．

〔リストB-4〕プログラム graph.asm ①

```
 1: ;*********************************************************************
 2: ;
 3: ;    機    能 :  グラフィック・ドライバ（メイン）
 4: ;    サ    ブ :  seg.h graphsub.asm graphc.c
 5: ;    生    成 :  make graph.mak
 6: ;    使用方法 :  DEVIVE = GRAPH
 7: ;
 8: ;*********************************************************************/
 9:            PAGE    60, 130
10:            .MODEL  SMALL, C
11:            INCLUDE seg.h
12:
13: GLIO_SIZE  EQU     1400h + 20h
14: STK_SIZE   EQU     2048
15: _acrtused  EQU     9876h
16:
17: ;*********************************************************************
18: ;
19: ;    機    能:    外部参照
20: ;
21: ;*********************************************************************/
22:            EXTRN   start:NEAR
23:            PUBLIC  d_end
24:            PUBLIC  glio_buff
25:            PUBLIC  _acrtused
26:            PUBLIC  entry, strategy
27:            PUBLIC  dev_header
28:
29: ;*********************************************************************
30: ;
31: ;    機    能:    ドライバの最終アドレスの定義
32: ;
33: ;*********************************************************************/
34: _dend      SEGMENT WORD PUBLIC 'BSS'
35: d_end      LABEL   WORD
36: _dend      ENDS
37:
38: ;*********************************************************************
39: ;
40: ;    SEGMENT:  コード・セグメント
41: ;    機    能 :  デバイス・ヘッダおよびプログラム／バッファ
42: ;
43: ;*********************************************************************/
44:            .CODE
45: ;*********************************************************************
46: ;
47: ;    機    能 :  デバイス・ヘッダ
48: ;
49: ;*********************************************************************/
50: dev_header LABEL   WORD
51:            DD      -1
52:            DW      8003h            ;キャラクタ・デバイス
53:            DW      strategy
54:            DW      entry
55:            DB      'CON     '       ;デバイス名
56:
57: ;*********************************************************************
58: ;
59: ;    機    能 :  GLIO用ワーク・エリア
60: ;
61: ;*********************************************************************/
62: glio_buff  DB      GLIO_SIZE dup (?)
63:
64: ;*********************************************************************
65: ;
66: ;    機    能 :  スタックの確保
67: ;
68: ;*********************************************************************/
69: stack      DB      STK_SIZE dup (?)
70: stk_btm    LABEL   WORD
71:
72: ;*********************************************************************
73: ;
74: ;    機    能:  バッファ
75: ;
```

〔リストB-4〕プログラム graph.asm ②

```
 76: ;************************************************************************/
 77: data_seg       DW      ?
 78: packet         DD      ?
 79: sp_buff        DW      ?
 80: ss_buff        DW      ?
 81:
 82: ;************************************************************************
 83: ;
 84: ;    ﾙｰﾁﾝ名 :   starategy
 85: ;    機  能 :   ストラテジ・ルーチン
 86: ;    入  力 :   ES:BX ･･･ コマンド・パケットへのポインタ
 87: ;    出  力 :   なし
 88: ;
 89: ;************************************************************************
 90: strategy       PROC    FAR
 91:                mov     WORD PTR cs:[packet], bx          ;コマンド・パケット格納
 92:                mov     WORD PTR cs:[packet + 2], es
 93:                ret
 94: strategy       ENDP
 95:
 96: ;************************************************************************
 97: ;
 98: ;    ﾙｰﾁﾝ名 :   entry
 99: ;    機  能 :   割り込みルーチン
100: ;    入  力 :   なし
101: ;    出  力 :   なし
102: ;
103: ;************************************************************************
104: entry          PROC    FAR USES AX BX CX DX SI DI BP DS
105:                push    es
106:                cli
107:                mov     cs:ss_buff, ss                    ;スタック退避
108:                mov     cs:sp_buff, sp
109:                mov     ax, cs
110:                mov     ds, ax
111:                mov     es, ax
112:                mov     ss, ax
113:                lea     sp, stk_btm
114:                push    WORD PTR packet + 2
115:                push    WORD PTR packet
116:                call    start
117:                add     sp, 4
118:                mov     sp, cs:sp_buff
119:                mov     ss, cs:ss_buff
120:                sti
121:                pop     es
122:                ret
123: entry          ENDP
124:                END
```

〔図B-3〕グラフ LIO アクセス時のレジスタ設定

これらのデータ定義が終われば，そのほかのスタックやコードの置かれるアドレスに関しては何も制約がないため，自由にプログラミングできることになります．同リストでは，ワーク・エリアの後にスタック・エリアを確保し，またその後にはGCドライバのワーク・エリアを確保しています．

プロシージャ strategy はストラテジ・ルーチンです．このプロシージャでは，ES：BX レジスタに渡されたコマンド・パケットへのポインタをGCドライバのワーク・エリアに格納してFARリターンしています．

プロシージャ entry は割り込みエントリ・ルーチンです．このプロシージャでは，SSレジスタとSPレジスタをGCドライバのワーク・エリアに格納したのち，デバイス・ドライバのベース・セグメント値をSS，ES，DSの各セグメント・レジスタに設定します．また，MS-DOSはデバイス・ドライバを呼び出す際に，スタック領域を20個程度しか用意していないため，SPレジスタをGCドライバのスタック領域に設定します．

次に，graphc.c モジュール内の関数 start を呼んで，ドライバに対するデバイス・コマンドの処理を行います．関数 start はC言語で記述されているため，コマンド・パケットへのFARポインタをスタックに積んでから関数 start をコールします(スタックを介して関数 start への引数として渡す)．

コマンド・パケットの解析や処理は関数 start が行い，制御がプロシージャ entry に返されたら，SS：SP レジスタ値をもとの値に復元して MS-DOS に FAR リターンします．

● struc.h

リストB-5 のヘッダ・ファイルは，graphc.c で取り込まれて参照されるファイルで，リクエスト・ヘッダやコマンド・パケットのデータ構造を定義しています．

構造体 _REQ_HEAD は，図8-5(283ページ)に示したリクエスト・ヘッダのデータ構造を定義しています．

構造体 _INIT_PACKET は，INIT コマンド用のコマンド・パケットのデータ構造を定義しています．

構造体 _RW_PACKET は，READ コマンドおよび WRITE コマンド用のコマンド・パケットのデータ構造を定義しています．

構造体 _NO_WAIT は，NON-DESTRUCTIVE READ コマンド用のコマンド・パケットのデータ構造を定義しています．

構造体 _DEV_HEAD は，デバイス・ヘッダのデータ構造を定義していて，オリジナル CON デバイス・ドライバを検索する際に用いられます．

● ASCII 制御コード ●

表C の ASCII 制御コードをコンソールに送ることにより，CRT 画面(カーソル)の制御を行うことができます．

▼〔表C〕
ASCII 制御コード

記号	16進数	機　　能
BEL	07	ブザーを1秒間鳴らす
BS	08	カーソルを1文字左に移動する
HT	09	カーソルを次のタブ位置(08, 16, 24, 32, 40, 48, 56, 64, 72)に移動する
LF	0A	カーソルを同じカラム位置で1行下へ移動する
VT	0B	カーソルを同じカラム位置で1行上へ移動する
FF	0C	カーソルを1文字右に移動する
CR	0D	カーソルを行の左端に移動する
SUB	1A	CRT 画面をすべてクリアする(カーソルはホーム位置となる)
ESC	1B	エスケープ・コード
RS	1E	カーソルをホーム位置に移動する

〔リストB-5〕
ヘッダ・ファイル
struc.h

```
 1: /*****************************************************************
 2: *
 3: *    機    能 :  構造体の定義
 4: *
 5: *****************************************************************/
 6: /*****************************************************************
 7: *
 8: *    構造体 :    _REQ_HEAD
 9: *    機    能 :   リクエスト・ヘッダの定義
10: *
11: *****************************************************************/
12: typedef struct _REQ_HEAD {
13:     unsigned char packet_len;    /* コマンド・パケットの長さ */
14:     unsigned char dev_code;      /* 論理装置コード */
15:     unsigned char cmd_code;      /* コマンド・コード */
16:     unsigned int  status;        /* ステータス */
17:     unsigned char reserve[8];    /* 予約域 */
18: } REQ_HEAD;
19:
20: /*****************************************************************
21: *
22: *    構造体 :     _INIT_PACKET
23: *    機    能 :   INIT コマンド用コマンド・パケットの定義
24: *
25: *****************************************************************/
26: typedef struct _INIT_PACKET {
27:     REQ_HEAD packet;             /* リクエスト・ヘッダ */
28:     unsigned char unit;          /* ユニット数 */
29:     char far *end;               /* エンド・アドレス */
30:     char far *bpb;               /* BPB配列へのポインタ (無視) */
31:     unsigned char dev_num;       /* ブロック・デバイス番号 (無視) */
32: } INIT_PACKET;
33:
34: /*****************************************************************
35: *
36: *    構造体 :     _RW_PACKET
37: *    機    能 :   READ & WRITE コマンド用コマンド・パケットの定義
38: *
39: *****************************************************************/
40: typedef struct _RW_PACKET {
41:     REQ_HEAD packet;               /* リクエスト・ヘッダ */
42:     unsigned char media_dsc;       /* メディア・ディスクリプタ */
43:              char far *trans;      /* 転送アドレス */
44:     unsigned int  bytes;           /* 転送バイト数 */
45:     unsigned int  sct_begin;       /* 開始セクタ (無視) */
46:              char far *id_ptr;     /* ボリュームIDへのポインタ (無視) */
47: } RW_PACKET;
48:
49: /*****************************************************************
50: *
51: *    構造体 :     _NOWAIT_PACKET
52: *    機    能 :   READ NO WAIT コマンド用コマンド・パケットの定義
53: *
54: *****************************************************************/
55: typedef struct _NOWAIT_PACKET {
56:     REQ_HEAD packet;               /* リクエスト・ヘッダ */
57:     unsigned char dev_data;        /* デバイスからのデータ */
58: } NOWAIT_PACKET;
59:
60: /*****************************************************************
61: *
62: *    構造体 :     _DEV_HEAD
63: *    機    能 :   デバイス・ヘッダの構造定義
64: *
65: *****************************************************************/
66: typedef struct _DEV_HEAD {
67:     struct _DEV_HEAD far *dev_link;
68:     int  dev_atr;
69:     int  strtgy;
70:     int  entry;
71:     char dev_name [8];
72: } DEV_HEAD;
```

● graphc.c モジュール

リストB-6のモジュールは，GCドライバのコマンド・コードの解析と処理を行います．

構造体_GRAPHTBLは，グラフLIOの割り込みベクタ番号とコマンド名の対応表のデータ構造を定義し，この構造体を用いて配列graph_cmdに実際のデータ(割り込みベクタ番号とグラフ制御コマンド名)が確保されます．

このファイルでは，グローバル変数名の最初の1文字に大文字を用いて，関数内で変数をアクセスする際にローカル変数とグローバル変数の区別が明確にできるようにしています．

関数startは，割り込みエントリ・ルーチンから渡されたコマンド・パケットへのポインタを用いて，各デバイス・コマンドの処理を行います．

まず，最初にコマンド・コードが0（INITコマンド）でなく，かつオリジナルCONデバイス・ドライバへのポインタCon_ptrがNULLかどうかを調べています．これらの条件が成立したらオリジナルCONデバイス・ドライバの検索を行います．

第8章で述べたように，デバイス・ヘッダにあるデバイス・リンク情報（先頭のダブル・ワード）は，MS-DOSによって設定されますが，INITコマンドが発行された時点で設定されるのではなく，INITコマンドの終了後に設定されます．また，オリジナルCONデバイス・ドライバを組み込んだあと，そのアドレスは固定されるため検索は一度行えばよく，ポインタCon_ptrが初期値（NULL）でなければすでに検索されたことになります．

よって，ここではオリジナルCONデバイス・ドライバの検索を，コマンド・コードがINITコマンド以外で，かつポインタCon_ptrがNULLのときに限って行っています．オリジナルCONデバイス・ドライバの検索は，GCドライバ自身のデバイス・ヘッダに設定されているリンク情報をもとに関数srch_devが行います．

オリジナルCONデバイス・ドライバの検索が終わったら，switch文を用いてコマンド・コードを調べ，各コマンド・コードに対応した関数を呼び出してデバイス・コマンドの処理を行っています．各関数からは，戻り値としてステータス・コードが返されるので，その値をコマンド・パケットのステータス・フィールドに設定して返します．もし，コマンド・コードが定義されているコード以外の場合は，ステータス・フィールドに8003H（無効なコマンド・コード）を返しています．

関数srch_devは，オリジナルCONデバイス・ドライバの検索を行います．この関数では，関数strcmpifnを用いてデバイス名がCONであるかどうかを調べ，一致していればそのデバイスへのポインタを戻り値として返します．もし，デバイス名が異なる場合は，そのデバイス・ヘッダのリンク情報をもとに，関数srch_devを再帰的に呼び出してオリジナルCONデバイス・ドライバが見つかるまで検索を行います．

関数initはINITコマンドの処理を行います．まず最初に，関数sdspを用いてGCドライバのタイトルを表示し，次にコマンド・パケットのブレーク・アドレス・フィールドにGCドライバの最終アドレス（ラベルd_endのアドレス）を設定して返します．

このあと，glio.cモジュール内の関数vect_initを用いてグラフLIOのベクタ設定を行い，グラフLIOへのアクセスを可能にしておきます．これらの処理が終わったら，戻り値としてDONEビットをセットしたステータス・コードを返します．

関数readは，READコマンドの処理を行います．READコマンドは，文字の入力を行うものであり，コンソール・ドライバは，キー入力された文字を割り込みを用いてバッファに格納しておき，このコマンド・コードが発行された時点でバッファ内の文字を返さなければなりません．この処理は，少々複雑なものとなり，また先に述べたBIOSのINT 18Hでは特殊なコードを返すため，この特殊コードからシフトJISコードへの変換も必要となります．ここでは，これらの処理をオリジナルCONデバイス・ドライバに任せることにし，関数con_callによって実現しています．

関数writeは，WRITEコマンドの処理を行うもので，グラフィック・コンソールの重要な処理を担当しています．この関数では，ESCコード（1BH）にベル・コード（07H）が続いていれば，グラフィック制御文字列とみなし，それに続く文字列を改行コードに出会うまで配列Str_buffに格納します．ここでは，このグラフィック制御文字列の開始コードのフラグとしてBgn_flagを使用しています．

実際のグラフィック制御は，関数glio_cmdに制御文字列へのポインタを渡して処理させています．グラフィック制御文字列でない場合は，関数cdspを用いて通常の文字列としてコンソールに表示します．すべての処理が終了したら戻り値としてステータス・コード（エラーなし）を返します．

関数no_waitは，NON-DESTRUCTIVE READコマンドの処理を行います．この関数では関数readと同様に，関数con_callを用いてオリジナルCONデバイス・ドライバにその処理を任せています．

関数read_statusは，READ STATUSコマンドの処理を行います．ここでは，関数（サブルーチン）chk_readを用いてキーボード・バッファの状態を調べ，その結果をステータス・コードとして返しています．

関数write_statusは，WRITE STATUSコマンドの処理を行います．実際には，この関数では何もすることはありません．ここでは，ステータス・コードとしてDONEビットのみをセットした0100H（エラーなし終了）を返しています．

関数flushは，READ FLUSHコマンドやWRITE

FLUSHコマンドの処理を行います．ここでも，実際には何もすることがないのでステータス・コードとして0100Hを返しています．

関数vect_initは，グラフLIO用のベクタの初期化を行います．すでに述べたように，グラフLIOのベクタは，セグメント・アドレスF990Hのオフセット0006Hから格納されているため，関数(サブルーチン)get_offを用いてグラフLIO処理ルーチンのオフセット・アドレスを読み出します．

そして，読み出されたオフセット・アドレスとグラフLIOのセグメント・アドレスF990Hを関数(サブルーチン)set_vectに渡してベクタの設定を行います．これらの設定は，割り込みベクタ番号A0HからAFHまで繰り返して行います．グラフLIOのベクタ設定が終わったら，関数(サブルーチン)set_c5を用いて割り込みベクタ番号C5Hのベクタ設定を行います．

これらの処理が終わったら，関数(サブルーチン)sys_callを用いてグラフLIOのGINITコマンドを実行し，グラフLIOの初期化を行います．次に，やはり関数sys_callによってグラフLIOのGCLSコマンドを実行して，グラフィック画面の初期化を行っておきます．

関数glio_cmdは，グラフィック制御文字列の解析と実行を行います．ここでは，渡されたグラフィック制御文字列に対して，do~whileループと関数strcmpiを用いて，グラフLIOのコマンド・テーブル内のコマンド名との比較を行い，glio.cモジュール内の対応する各関数を呼び出してグラフィック処理を行っています．

関数get_symbolは，グラフィック制御文字列の中から，一つのシンボル(コマンド名やパラメータとなる数字)を切り出し，バッファSym_buffに格納します．このとき，グラフィック制御文字列の中にまだシンボルが残っていれば偽(ゼロ：FALSE)を返し，もし改行コードしか残っていない場合は真(ゼロ以外：TRUE)を返します．

関数get_valは，関数get_symbolを用いてグラフィック制御文字列の中から一つのシンボルを切り出し，そのシンボルを数字とみなして関数atoiを用いて数値に変換します．変換された数値はグローバル変数Valに格納するとともに戻り値としても返しています．

関数strcmpiは，二つのポインタで指定された文字列を比較する関数で，MS-C標準の同名の関数と同等の処理を行います．この関数では，二つの文字列をすべて小文字に変換して比較することによって大文字と小文字の区別をなくしています．

比較の結果，二つの文字列が等しい場合は"0"(FALSE)を，文字列1が文字列2よりも文字コードの大きいものを含んでいる場合には"1"(TRUE)を返しています．また，文字列1が文字列2よりも文字コードの小さいものを含んでいる場合には"−1"を返しています．

関数strcmpifnは，関数strcmpiと同等の処理を行いますが，文字列の長さを指定できるため，文字列がASCIZ文字列でなくてもかまいません．また，この関数では文字列へのポインタがFARポインタでなければなりません．

関数atoiは，ポインタで渡された文字列を数値に変換します．変換された数値(ワード値)は戻り値として返されます．

関数isdigitは，渡された文字コードが10進数字の文字コードであるかどうかを調べます．もし，10進数字の場合は真(ゼロ以外：TRUE)を返し，それ以外の文字コードの場合は偽(ゼロ：FALSE)を返します．

関数tolowerは，渡された文字を小文字に変換し，その結果を戻り値として返します．

〔リストB-6〕プログラムgraphc.c ①

```
 1: /*********************************************************************
 2: *
 3: *    機    能:   グラフィック・ドライバ(コマンド部分)
 4: *    生    成:   cl -AS -c graphc.c
 5: *
 6: ********************************************************************/
 7: #include        "struc.h"
 8: #define         TRUE            1
 9: #define         FALSE           0
10: #define         PLUS            1
11: #define         ZERO            0
12: #define         MINUS           -1
13: #define         NULL            0
14: #define         NO_ERR          0x0100
15: #define         ERR_CMD         0x8003
16: #define         ERR_ETC         0x800C
17: #define         BUSY            0x0300
18: #define         STR_LEN         10
19: #define         BUFF_LEN        256
20: #define         LIO_SEG         0xF990          /* グラフLIOのセグメント */
```

〔リストB-6〕プログラム graphc.c ②

```
21:
22: /*********************************************************************
23: *
24: *    機  能:  グラフＬＩＯコマンド・テーブルの定義
25: *
26: *********************************************************************/
27: typedef struct _GRAPHTBL {
28:     int vect;
29:     char *cmd;
30: } GRAPHTBL;
31:
32: /*********************************************************************
33: *
34: *    機  能:  グラフＬＩＯコマンド・テーブルの確保と初期化
35: *
36: *********************************************************************/
37: GRAPHTBL graph_cmd [] = {
38:     0xA0,   "init",
39:     0xA1,   "screen",
40:     0xA2,   "view",
41:     0xA3,   "color1",
42:     0xA4,   "color2",
43:     0xA5,   "cls",
44:     0xA6,   "pset",
45:     0xA7,   "line",
46:     0xA8,   "circle",
47:     0xA9,   "paint1",
48:     0xAA,   "paint2",
49:     0xAB,   "get",
50:     0xAC,   "put1",
51:     0xAD,   "put2",
52:     0xAE,   "roll",
53:     0xAF,   "point2",
54:     0x00,   ""
55: };
56:
57: /*********************************************************************
58: *
59: *    機  能:  グローバル変数の宣言
60: *
61: *********************************************************************/
62: int Bgn_flag = FALSE;
63: int Esc_flag = FALSE;
64: int Cr_flag = FALSE;
65: char Str_buff [BUFF_LEN];
66: char Sym_buff [BUFF_LEN];
67: char *Str_ptr;
68: DEV_HEAD far *Con_ptr = NULL;
69:
70: /*********************************************************************
71: *
72: *    機  能:  ＡＮＳＩプロトタイプ関数宣言
73: *
74: *********************************************************************/
75: void start (REQ_HEAD far *);
76: DEV_HEAD far *srch_dev (DEV_HEAD far *);
77: int  init (INIT_PACKET far *);
78: int  read (RW_PACKET far *);
79: int  write (RW_PACKET far *);
80: int  no_wait (NOWAIT_PACKET far *);
81: int  read_status (REQ_HEAD far *);
82: int  write_status (REQ_HEAD far *);
83: int  flush (REQ_HEAD far *);
84: void vect_init (void);
85: int  glio_cmd (char *);
86: int  get_symbol (void);
87: int  get_val (void);
88: int  atoi (char *);
89: int  isdigit (int);
90: int  strcmpi (char *, char *);
91: int  strcmpifn (char far *, char far *, int);
92: int  tolower (int);
93: void con_call (DEV_HEAD far *, REQ_HEAD far *);
94: int  get_off (int, int);
95: int  get_ds (void);
96: void set_vect (int, int, int);
97: int  sys_call (int, char *);
98: void set_c5 (void);
99: int  chk_read (void);
```

〔リストB-6〕プログラム graphc.c ③

```
100: void sdsp (char *);
101: void cdsp (int);
102:
103: /*************************************************************
104: *
105: *    関数名：   start ()
106: *    機　能：   コマンド・コードの解析
107: *    入　力：   ptr ・・・ コマンド・パケットへのポインタ
108: *    出　力：   なし
109: *
110: ***************************************************************/
111: void start (ptr)
112: REQ_HEAD far *ptr;
113: {
114:     int ret_code;
115:     extern DEV_HEAD dev_header;
116:     DEV_HEAD far *hptr;
117:
118:     if (ptr -> cmd_code && !Con_ptr) {          /* オリジナルＣＯＮデバイス検索 */
119:         hptr = (DEV_HEAD far *)&dev_header;
120:         Con_ptr = srch_dev (hptr -> dev_link);
121:     }
122:     switch (ptr -> cmd_code) {                  /* デバイス・コマンド */
123:     case 0:                                      /* READ */
124:         ret_code = init ((INIT_PACKET far *)ptr); break;
125:     case 4:                                      /* WRITE */
126:         ret_code = read ((RW_PACKET far *)ptr); break;
127:     case 5:                           /* Non-destructive READ No-wait */
128:         ret_code = no_wait ((NOWAIT_PACKET far *)ptr); break;
129:     case 6:                                      /* Read Status */
130:         ret_code = read_status ((REQ_HEAD far *)ptr); break;
131:     case 7:                                      /* READ FLUSH */
132:         ret_code = flush ((REQ_HEAD far *)ptr); break;
133:     case 8:                                      /* WRITE */
134:     case 9:                                      /* WRITE & VERIFY */
135:         ret_code = write ((RW_PACKET far *)ptr); break;
136:     case 10:                                     /* WRITE STATUS */
137:         ret_code = write_status ((REQ_HEAD far *)ptr); break;
138:     case 11:                                     /* WRITE FLUSH */
139:         ret_code = flush ((REQ_HEAD far *)ptr); break;
140:     default:
141:         ptr -> status = ERR_CMD;
142:         return;
143:     }
144:     ptr -> status = ret_code;
145: }
146:
147: /*************************************************************
148: *
149: *    関数名     srch_dev (ptr)
150: *    機　能：   オリジナルＣＯＮデバイスの検索
151: *    入　力：   ptr ・・・・ デバイス・ヘッダへのポインタ
152: *    出　力：   ＣＯＮデバイス・ヘッダへのポインタ
153: *               別のデバイスの場合： NULL
154: *
155: ***************************************************************/
156: DEV_HEAD far *srch_dev (ptr)
157: DEV_HEAD far *ptr;
158: {
159:     DEV_HEAD far *link;
160:
161:     if (strcmpifn (ptr -> dev_name, (char far *)"CON     ", 8)) {
162:         if (link = srch_dev (ptr -> dev_link)){
163:             return (link);
164:         } else {
165:             return (NULL);
166:         }
167:     }
168:     return (ptr);
169: }
170:
171: /*************************************************************
172: *
173: *    関数名：   init (ptr)
174: *    機　能：   INIT コマンド
175: *    入　力：   ptr ・・・・ コマンド・パケットへのポインタ
176: *    出　力：   ステータス・コード
177: *
178: ***************************************************************/
```

〔リストB-6〕プログラム graphc.c ④

```
179: int init (ptr)
180: INIT_PACKET far *ptr;
181: {
182:     extern int d_end;
183:
184:     sdsp ("グラフィック・コンソール・ドライバ");
185:     sdsp ("  Ver.1.1 for PC9801  programed by H.Abe¥n");
186:     ptr -> end = (char far *)&d_end;                    /* 最終ドレス */
187:     vect_init ();
188:     return (NO_ERR);
189: }
190:
191: /********************************************************************
192: *
193: *    関数名 :   read (ptr)
194: *    機  能 :   READ コマンド
195: *    入  力 :   ptr ・・・ コマンド・パケットへのポインタ
196: *    出  力 :   ステータス・コード
197: *
198: ********************************************************************/
199: int read (ptr)
200: RW_PACKET far *ptr;
201: {
202:
203:     con_call (Con_ptr, (REQ_HEAD far *)ptr);
204:     return (ptr -> packet.status);
205: }
206:
207: /********************************************************************
208: *
209: *    関数名 :   write (ptr)
210: *    機  能 :   READ コマンド
211: *    入  力 :   ptr ・・・ コマンド・パケットへのポインタ
212: *    出  力 :   ステータス・コード
213: *
214: ********************************************************************/
215: int write (ptr)
216: RW_PACKET far *ptr;
217: {
218:     int ret_code = NO_ERR;
219:     int c;
220:
221:     if (! Bgn_flag) {
222:         Str_ptr = Str_buff;
223:     }
224:     c = *ptr ->trans;
225:     if (c == 0x1B) {
226:         Esc_flag = TRUE;
227:         return (ret_code);
228:     }
229:     if (c == 0x07 && Esc_flag) {
230:         Bgn_flag = TRUE;
231:         return (ret_code);
232:     }
233:     if (! Bgn_flag && Esc_flag) {
234:         Esc_flag = FALSE;
235:         cdsp (0x1B);
236:     }
237:     if (Bgn_flag) {
238:         *Str_ptr ++ = c;
239:         if (c == 0x0D) {
240:             Cr_flag = TRUE;
241:             return (ret_code);
242:         }
243:         if (Cr_flag && c == 0x0A) {
244:             ret_code = glio_cmd (Str_buff);
245:             Bgn_flag = FALSE;
246:             Esc_flag = FALSE;
247:             Cr_flag = FALSE;
248:         }
249:     } else {
250:         cdsp (c);
251:     }
252:     return (ret_code);
253: }
254:
```

〔リストB-6〕プログラム graphc.c ⑤

```
255: /*********************************************************************
256: *
257: *    関数名:   no_wait (ptr)
258: *    機  能:   非破壊読み込み
259: *    入  力:   ptr ・・・・ コマンド・パケットへのポインタ
260: *    出  力:   ステータス・コード
261: *
262: *********************************************************************/
263: int no_wait (ptr)
264: NOWAIT_PACKET far *ptr;
265: {
266:     con_call (Con_ptr, (REQ_HEAD far *)ptr);
267:     return (ptr -> packet.status);
268: }
269:
270: /*********************************************************************
271: *
272: *    関数名:   read_status (ptr)
273: *    機  能:   READ STATUS コマンド
274: *    入  力:   ptr ・・・・ コマンド・パケットへのポインタ
275: *    出  力:   ステータス・コード
276: *
277: *********************************************************************/
278: int read_status (ptr)
279: REQ_HEAD far *ptr;
280: {
281:     int ret_code;
282:
283:     if (chk_read ()) {
284:         ret_code = NO_ERR;
285:     } else {
286:         ret_code = BUSY;
287:     }
288: }
289:
290: /*********************************************************************
291: *
292: *    関数名:   write_status (ptr)
293: *    機  能:   OUTPUT STATUS コマンド
294: *    入  力:   ptr ・・・・ コマンド・パケットへのポインタ
295: *    出  力:   ステータス・コード
296: *
297: *********************************************************************/
298: int write_status (ptr)
299: REQ_HEAD far *ptr;
300: {
301:     return (NO_ERR);
302: }
303:
304: /*********************************************************************
305: *
306: *    関数名:   flush (ptr)
307: *    機  能:   READ/WRITE FLUSH コマンド
308: *    入  力:   ptr ・・・・ コマンド・パケットへのポインタ
309: *    出  力:   ステータス・コード
310: *
311: *********************************************************************/
312: int flush (ptr)
313: REQ_HEAD far *ptr;
314: {
315:     return (NO_ERR);
316: }
317:
318: /*********************************************************************
319: *
320: *    関数名:   vect_init ()
321: *    機  能:   グラフL I O用ベクタの初期化
322: *    入  力:   なし
323: *    出  力:   なし
324: *
325: *********************************************************************/
326: void vect_init ()
327: {
328:     unsigned int off;
329:     char buff [1];                  /* ダミー */
330:     int i;
331:
```

〔リストB-6〕プログラム graphc.c ⑥

```
332:     for (i = 0; i < 16; i ++) {
333:         off = get_off (i * 4 + 6, LIO_SEG) ;
334:         set_vect (0xA0 + i, LIO_SEG, off);
335:     }
336:     set_c5 ();
337:     sys_call (0xA0, buff);              /* GINIT */
338:     sys_call (0xA5, buff);              /* GCLS */
339: }
340:
341: /*****************************************************************
342: *
343: *   関数名 :   glio_cmd (ptr)
344: *   機  能 :   グラフＬＩＯコマンドの解析と実行
345: *   入  力 :   ptr ・・・・・ 文字へのポインタ
346: *   出  力 :   終了ステータス
347: *
348: *****************************************************************/
349: int glio_cmd (ptr)
350: char *ptr;
351: {
352:     GRAPHTBL *tbl = graph_cmd;
353:
354:     Str_ptr = Str_buff;
355:     get_symbol ();
356:     do {
357:         if (! strcmpi (Sym_buff, tbl -> cmd)) {
358:             switch (tbl -> vect) {
359:             case 0xA1:
360:                 screen (ptr); break;
361:             case 0xA2:
362:                 view (ptr); break;
363:             case 0xA3:
364:                 color1 (ptr); break;
365:             case 0xA4:
366:                 color2 (ptr); break;
367:             case 0xA5:
368:                 cls (ptr); break;
369:             case 0xA6:
370:                 pset (ptr); break;
371:             case 0xA7:
372:                 line (ptr); break;
373:             case 0xA8:
374:                 circle (ptr); break;
375:             case 0xA9:
376:                 paint1 (ptr); break;
377:             case 0xAA:
378:                 paint2 (ptr); break;
379:             case 0xAB:
380:                 get (ptr); break;
381:             case 0xAC:
382:                 put1 (ptr); break;
383:             case 0xAD:
384:                 put2 (ptr); break;
385:             case 0xAE:
386:                 roll (ptr); break;
387:             case 0xAF:
388:                 point2 (ptr); break;
389:             default:
390:                 return (ERR_ETC);                          /* 一般的なエラー */
391:             }
392:             return (NO_ERR);                               /* 正常終了 */
393:         }
394:         tbl ++;
395:     } while (tbl -> vect);
396:     return (ERR_ETC);                                      /* 一般的なエラー */
397: }
398:
399: /*****************************************************************
400: *
401: *   関数名 :   get_symbol (ptr)
402: *   機  能 :   シンボルの切り出し
403: *   入  力 :   ptr ・・・・・ 文字へのポインタ
404: *   出  力 :   FALSE(00H): 正常終了  TRUE(01H): 改行コードの検出
405: *              Sym_buff ・・・・・ シンボル
406: *
407: *****************************************************************/
408: int get_symbol ()
```

〔リストB-6〕プログラム graphc.c ⑦

```
409: {
410:     char *sym_ptr = Sym_buff;
411:     int c;
412:
413:     if (*Str_ptr == 0x0A) {
414:         return (TRUE);
415:     }
416:     while ((c = *Str_ptr) == ' ' || c == ',') {
417:         Str_ptr ++;
418:     }
419:     while (! ((c = *Str_ptr++) == ' ' || c == ',' || c == 0x0D || c == 0x0A)) {
420:         *sym_ptr++ = c;
421:     }
422:     *sym_ptr = '¥0';
423:     return (FALSE);
424: }
425:
426: /****************************************************************
427: *
428: *    関数名：   get_val ()
429: *    機  能：   数字の切り出しと変換
430: *    入  力：   なし
431: *    出  力：   00H: 正常終了    01H: 改行コードの検出
432: *               Val ・・・・ 数値
433: *
434: ****************************************************************/
435: int get_val ()
436: {
437:     if (get_symbol ()) {
438:         return (ZERO);
439:     }
440:     return (atoi (Sym_buff));
441: }
442:
443: /****************************************************************
444: *
445: *    関数名：   strcmpi (s1, s2)
446: *    機  能：   文字列の比較
447: *    入  力：   s1 ・・・・ 文字列へのポインタ
448: *               s2 ・・・・ 文字列へのポインタ
449: *    出  力：   s1 > s2 : TRUE (1)
450: *               s1 = s2 : FALSE (0)
451: *               s1 < s2 : MINUS (-1)
452: *
453: ****************************************************************/
454: int strcmpi (s1, s2)
455: char *s1, *s2;
456: {
457:     while (*s1 != '¥0' || *s2 != '¥0') {
458:         if (tolower (*s1) > tolower (*s2)) {
459:             return (PLUS);
460:         }
461:         if (tolower (*s1) < tolower (*s2)) {
462:             return (MINUS);
463:         }
464:         s1 ++;
465:         s2 ++;
466:     }
467:     return (ZERO);
468: }
469:
470: /****************************************************************
471: *
472: *    関数名：   strcmpifn (s1, s2)
473: *    機  能：   文字列の比較
474: *    入  力：   s1 ・・・・ 文字列へのFARポインタ
475: *               s2 ・・・・ 文字列へのFARポインタ
476: *               n  ・・・・ 比較する文字数
477: *    出  力：   s1 > s2 : TRUE (1)
478: *               s1 = s2 : FALSE (0)
479: *               s1 < s2 : MINUS (-1)
480: *
481: ****************************************************************/
482: int strcmpifn (s1, s2, n)
483: char far *s1, far *s2;
484: int n;
485: {
```

〔リストB-6〕プログラム graphc.c ⑧

```
486:     do {
487:         if (tolower (*s1) > tolower (*s2)) {
488:             return (PLUS);
489:         }
490:         if (tolower (*s1) < tolower (*s2)) {
491:             return (MINUS);
492:         }
493:         s1 ++;
494:         s2 ++;
495:     } while (--n);
496:     return (ZERO);
497: }
498:
499: /*************************************************************
500: *
501: *    関数名 :    atoi (s)
502: *    機  能 :    文字列 → int への変換
503: *    入  力 :    s ・・・・・ 文字列へのポインタ
504: *    出  力 :    ワード整数
505: *
506: *************************************************************/
507: int atoi(s)
508: char *s;
509: {
510:     int sign;
511:     unsigned int i;
512:
513:     while (*s == ' ') {
514:         s++;
515:     }
516:     sign = 1;
517:     switch (*s) {
518:     case '-':
519:         sign = -1;
520:     case '+':
521:         s++;
522:         break;
523:     }
524:     for (i = 0; isdigit (*s); s++) {
525:         i = 10 * i + (*s - '0');
526:     }
527:     return (sign * i);
528: }
529:
530: /*************************************************************
531: *
532: *    関数名 :    isdigit (c)
533: *    機  能 :    10進数字のチェック
534: *    入  力 :    s ・・・・・ 文字コード
535: *    出  力 :    10進数字 : TRUE        それ以外 : FALSE
536: *
537: *************************************************************/
538: int isdigit (c)
539: int c;
540: {
541:     if (c >= '0' && c <= '9') {
542:         return (TRUE);
543:     }
544:     return (FALSE);
545: }
546:
547: /*************************************************************
548: *
549: *    関数名 :    tolower (c)
550: *    機  能 :    小文字への変換
551: *    入  力 :    c ・・・・・ 文字コード
552: *    出  力 :    小文字のコード
553: *
554: *************************************************************/
555: int tolower (c)
556: int c;
557: {
558:     if (c >= 'A' && c <= 'Z') {
559:         return (c - ('a' - 'A'));
560:     }
561:     return (c);
562: }
```

● glio.c モジュール

リストB-7のモジュールには，グラフLIOのアクセスを行うための関数が記述されています．

関数gscreenは，与えられた四つのパラメータをバッファに設定し，そのバッファへのポインタを関数sys_callに渡してINT A1Hを実行してグラフィック画面モードの設定を行います．

関数gviewは，与えられた6個のパラメータをバッファに設定し，そのバッファへのポインタを関数sys_callに渡してINT A2Hを実行してグラフィック描画領域の指定を行います．ここで，第1パラメータ～第4パラメータは，ワード値で設定しなければなりません．

C言語において，ワード値とバイト値が混合したバッファを確保する場合，

```
int   buff_w[];
char buff_b[];
```

のように宣言して領域を確保することも考えられます．しかし，この方法ではC処理系によって必ずしも連続した領域が確保されるという保証はありません．

このために，ここではバッファとしてバイト配列の確保を行い，その該当する位置(アドレス)にワード値を格納する方法として，ワード値へのキャストを行うことによって，ワードとバイトのミックスされたバッファを連続した領域に確保しています．

関数gcolor1は，与えられた三つのパラメータをバッファに設定し，そのバッファへのポインタを関数sys_callに渡してINT A3Hを実行して背景色の指定を行います．

関数gcolor2は，与えられた二つのパラメータをバッファに設定し，そのバッファへのポインタを関数sys_callに渡してINT A4Hを実行してパレット番号と表示色コードの対応を指定します．

関数glcsは，関数sys_callを用いてINT A5Hを実行してグラフィック画面のクリアを行います．

関数gpsetは，与えられた三つのパラメータをバッファに設定し，そのバッファへのポインタを関数sys_callに渡してINT A6Hを実行してドットの描画を行います．

関数glineは，与えられたパラメータをバッファに設定し，そのバッファへのポインタを関数sys_callに渡してINT A7Hを実行して直線や矩形の描画を行います．

すでに述べたように，グラフLIOのGLINEコマンドでは，ライン・スタイル・スイッチによって，パラメータのもつ意味が異なります．ここでは，このライン・スタイル・スイッチによる機能(バッファ位置)の違いをswitch文によって処理しています．

関数gcircleは，与えられたパラメータをバッファに設定し，そのバッファへのポインタを関数sys_call

〔リストB-7〕プログラム glio.c ①

```
 1: /*****************************************************************
 2: *
 3: *    機    能:   グラフィックＬＩＯのアクセス
 4: *    生    成:   cl -AS -c glio.c
 5: *
 6: ******************************************************************/
 7: #define WORK_SIZE     100
 8: #define DS_OFF        0x20
 9:
10: /*****************************************************************
11: *
12: *    機  能:   ＡＮＳＩプロトタイプ関数宣言
13: *
14: ******************************************************************/
15: void screen (void);
16: void view (void);
17: void color1 (void);
18: void color2 (void);
19: void cls (void);
20: void pset (void);
21: void line (void);
22: void paint1 (void);
23: void paint2 (void);
24: void get (void);
25: void put1 (void);
26: void put2 (void);
27: void roll (void);
28: void point2 (void);
29:
30: /*****************************************************************
31: *
32: *    関数名:   screen ()
33: *    機  能:   グラフィック画面モードの指定
```

に渡してINT A8Hを実行して円や楕円の描画を行います．グラフLIOのGCIRCLEコマンドは，同GLINEコマンドと同様に，フラグによってパラメータのもつ意味が異なります．ここでは，このフラグによる機能の違いをif文によって処理しています．

関数gpaint1は，与えられた四つのパラメータをバッファに設定し，そのバッファへのポインタを関数sys_callに渡してINT A9Hを実行して，指定された領域の塗りつぶしを指定された色で行います．ここで，グラフLIOに対してワーク領域を確保し，そのオフセットをパラメータ・バッファに設定してやらなければなりません．

また，ここではDSレジスタが20Hバイト分だけオフセットしていることに注意しなければなりません．これに伴いパラメータに与えるワーク領域のオフセット・アドレスも20Hバイト分だけオフセットして設定します．

関数gpaint2は，与えられたパラメータをバッファに設定し，そのバッファへのポインタを関数sys_callに渡してINT ABHを実行して，指定された領域の塗りつぶしを指定されたタイル・パターンで行います．ここで，グラフLIOに対してワーク領域を確保し，そのオフセットをパラメータ・バッファに設定していますが，関数gpaint1と同様に，パラメータに与えるワーク領域のオフセット・アドレスは20Hバイト分だけオフセットして設定します．

関数ggetは，与えられた七つのパラメータをバッファに設定し，そのバッファへのポインタを関数sys_callに渡してINT AAHを実行してドット情報の格納を行います．

関数gput1は，与えられたパラメータをバッファに設定し，そのバッファへのポインタを関数sys_callに渡してINT ACHを実行して格納されたドット情報の表示を行います．ここで，カラー・スイッチによってパラメータのもつ機能が異なるため，if文によって処理の違いを判断しています．

関数gput2は，与えられたパラメータをバッファに設定し，そのバッファへのポインタを関数sys_callに渡してINT ADHを実行して日本語の表示を行います．ここでも，カラー・スイッチによってパラメータのもつ機能が異なるため，if文によって処理の違いを判断しています．

関数grollは，与えられた三つのパラメータをバッファに設定し，そのバッファへのポインタを関数sys_callに渡してINT AEHを実行してグラフィック画面の移動を行います．

関数point2は，与えられた二つのパラメータをバッファに設定し，そのバッファへのポインタを関数sys_callに渡してINT AFHを実行してドットに対するパレット番号の通知を行います．

〔リストB-7〕プログラムglio.c ②

```
34: *    入  力:   int p1(mode)  ･･･ 画面モード
35: *              int p2(sw)    ･･･ スイッチ
36: *              int p3(act)   ･･･ アクティブ画面
37: *              int p4(dsp)   ･･･ ディスプレイ画面
38: *    出  力:   なし
39: *
40: *****************************************************************************/
41: void screen ()
42: {
43:     char buff [4];
44:
45:     buff [0] = get_val ();
46:     buff [1] = get_val ();
47:     buff [2] = get_val ();
48:     buff [3] = get_val ();
49:     sys_call (0xA1, buff);
50: }
51:
52: /****************************************************************************
53: *
54: *    関数名:   view ()
55: *    機  能:   描画領域の指定
56: *    入  力:   int p1(x1)    ･･･ 左上X座標
57: *              int p2(y1)    ･･･ 左上Y座標
58: *              int p3(x2)    ･･･ 右下X座標
59: *              int p4(y2)    ･･･ 右下Y座標
60: *              int p5(col1)  ･･･ 領域色
61: *              int p6(col2)  ･･･ 境界色
62: *    出  力:   なし
63: *
64: *****************************************************************************/
65: void view ()
66: {
```

〔リストB-7〕プログラム glio.c ③

```
 67:     char buff [9];
 68:
 69:     *(int *)(buff + 0) = get_val ();    /* 左上Ｘ座標 */
 70:     *(int *)(buff + 2) = get_val ();    /* 左上Ｙ座標 */
 71:     *(int *)(buff + 4) = get_val ();    /* 右下Ｘ座標 */
 72:     *(int *)(buff + 6) = get_val ();    /* 右下Ｙ座標 */
 73:     *(buff + 8) = get_val ();           /* 領域色 */
 74:     *(buff + 9) = get_val ();           /* 境界色 */
 75:     sys_call (0xA2, buff);
 76: }
 77:
 78: /*****************************************************************
 79: *
 80: *    関数名 :    color1 ()
 81: *    機　能 :    背景色の指定
 82: *    入　力 :    int p1(col1) ･･･ バックグランド
 83: *                int p2(col2) ･･･ ボーダー
 84: *                int p3(col3) ･･･ フォアグランド
 85: *    出　力 :    なし
 86: *
 87: *****************************************************************/
 88: void color1 ()
 89: {
 90:     char buff [4];
 91:
 92:     *(buff + 1) = get_val ();           /* バック・グランド色 */
 93:     *(buff + 2) = get_val ();           /* ボーダー色 */
 94:     *(buff + 3) = get_val ();           /* フォア・グランド色 */
 95:     sys_call (0xA3, buff);
 96: }
 97:
 98: /*****************************************************************
 99: *
100: *    関数名 :    color2 ()
101: *    機　能 :    パレット番号と表示色コードの対応
102: *    入　力 :    int p1(pal) ･･･ パレット番号
103: *                int p2(col) ･･･ 表示色
104: *    出　力 :    なし
105: *
106: *****************************************************************/
107: void color2 ()
108: {
109:     char buff [2];
110:
111:     *(buff + 0) = get_val ();           /* パレット番号 */
112:     *(buff + 1) = get_val ();           /* 表示色 */
113:     sys_call (0xA4, buff);
114: }
115:
116: /*****************************************************************
117: *
118: *    関数名 :    cls ()
119: *    機　能 :    描画領域の塗りつぶし
120: *    入　力 :    なし
121: *    出　力 :    なし
122: *
123: *****************************************************************/
124: void cls()
125: {
126:     sys_call (0xA5);
127: }
128:
129: /*****************************************************************
130: *
131: *    関数名 :    pset ()
132: *    機　能 :    点の描画
133: *    入　力 :    int p1(x)     ･･･ Ｘ座標
134: *                int p2(y)     ･･･ Ｙ座標
135: *                int p3(col)   ･･･ 表示色
136: *                int p4(mode)  ･･･ モード
137: *    出　力 :    なし
138: *
139: *****************************************************************/
140: void pset ()
141: {
142:     char buff [5];
143:
```

〔リストB-7〕プログラム glio.c ④

```
144:      *(int *)(buff + 0) = get_val ();          /* X座標 */
145:      *(int *)(buff + 2) = get_val ();          /* Y座標 */
146:      *(buff + 4) = get_val ();                 /* パレット番号 */
147:      sys_call (0xA6, buff);
148: }
149:
150: /*********************************************************************
151:  *
152:  *    関数名 :   line ()
153:  *    機  能 :   直線|矩形の描画
154:  *    入  力 :   int p1(x1)    ・・・始点X座標
155:  *               int p2(y1)    ・・・始点Y座標
156:  *               int p3(x2)    ・・・終点X座標
157:  *               int p4(y2)    ・・・終点Y座標
158:  *               int p5(pal1)  ・・・境界色
159:  *               int p6(code)  ・・・スイッチ
160:  *               int p6(sw)    ・・・スイッチ
161:  *               int p7(a1)    ・・・領域色|ラインスタイル|タイル長
162:  *               int p8(a2)    ・・・格納域オフセット
163:  *               int p9(a3)    ・・・格納域セグメント
164:  *    出  力:    なし
165:  *
166:  *********************************************************************/
167: void line ()
168: {
169:      int a1, a2, a3;
170:      char buff [18];
171:
172:      *(int *)(buff + 0) = get_val ();          /* 始点のX座標 */
173:      *(int *)(buff + 2) = get_val ();          /* 始点のY座標 */
174:      *(int *)(buff + 4) = get_val ();          /* 終点のX座標 */
175:      *(int *)(buff + 6) = get_val ();          /* 終点のY座標 */
176:      *(buff + 8) = get_val ();                 /* パレット番号*/
177:      *(buff + 9) = get_val ();                 /* 描画指定 */
178:      *(buff + 10) = get_val ();                /* ライン・スタイル・スイッチ */
179:      a1 = get_val ();
180:      a2 = get_val ();
181:      a3 = get_val ();
182:      switch (*(buff + 10)){                           /* スイッチ */
183:      case 0:                                   /* 指定なし */
184:          break;
185:      case 1:                                   /* ラインスタイル or 塗りつぶし */
186:          if (*(buff + 9) == 2) {               /* 矩形塗りつぶし */
187:              *(buff + 11) = a1;                /* 塗りつぶし色 */
188:          } else {                              /* ラインスタイル */
189:              *(int *)(buff + 11) = a1;         /* ライン・スタイル */
190:          }
191:          break;
192:      case 2:                                   /* タイル・パターン */
193:          *(buff + 13) = a1;                    /* タイルパターン長 */
194:          *(int *)(buff + 14) = a2;             /* 格納域オフセット */
195:          *(int *)(buff + 16) = a3;             /* 格納域アドレス */
196:          break;
197:      }
198:      sys_call (0xA7, buff);
199: }
200:
201: /*********************************************************************
202:  *
203:  *    関数名 :   circle ()
204:  *    機  能 :   円|楕円の描画
205:  *    入 力 :    int p1(cx)     ・・・中心X座標
206:  *               int p2(cy)     ・・・中心Y座標
207:  *               int p3(rx)     ・・・半径X座標
208:  *               int p4(ry)     ・・・半径Y座標
209:  *               int p5(pal)    ・・・境界色
210:  *               int p6(flg)    ・・・フラグ
211:  *               int p7(a1)     ・・・領域色|タイル長|開始点座標x1
212:  *               int p8(a2)     ・・・格納域オフセット|開始点座標y1
213:  *               int p9(a3)     ・・・格納域セグメント|終了点座標x2
214:  *               int p10(a4)    ・・・終了点座標y2
215:  *               int p11(a5)    ・・・領域色|タイル長
216:  *               int p12(a6)    ・・・格納域オフセット
217:  *               int p13(a7)    ・・・格納域セグメント
218:  *    出  力 :   なし
219:  *
```

〔リストB-7〕プログラム glio.c ⑤

```
220: ************************************************************************/
221: void circle ()
222: {
223:     int a1, a2, a3, a4, a5, a6, a7;
224:     char buff [23];
225:
226:     *(int *)(buff + 0) = get_val ();        /* 中心点X座標 */
227:     *(int *)(buff + 2) = get_val ();        /* 中心点Y座標 */
228:     *(int *)(buff + 4) = get_val ();        /* 半径X座標 */
229:     *(int *)(buff + 6) = get_val ();        /* 半径Y座標 */
230:     *(buff + 8) = get_val ();               /* パレット番号 */
231:     *(buff + 9) = get_val ();               /* フラグ */
232:     a1 = get_val ();
233:     a2 = get_val ();
234:     a3 = get_val ();
235:     a4 = get_val ();
236:     a5 = get_val ();
237:     a6 = get_val ();
238:     a7 = get_val ();
239:     if (*(buff + 9) & 0x03){                /* 開始点，終了点指定あり */
240:         *(int *)(buff + 10) = a1;           /* 開始点X座標 */
241:         *(int *)(buff + 12) = a2;           /* 開始点Y座標 */
242:         *(int *)(buff + 14) = a3;           /* 終了点X座標 */
243:         *(int *)(buff + 16) = a4;           /* 終了点Y座標 */
244:         if (*(buff + 9) & 0x40){            /* タイル・パターン指示あり */
245:             *(buff + 18) = a5;              /* タイル・パターン長 */
246:             *(int *)(buff + 19) = a6;       /* 格納域オフセット */
247:             *(int *)(buff + 21) = a7;       /* 格納域セグメント */
248:         } else if (*(buff + 9) & 0x20) {    /* 塗りつぶし指示 */
249:             *(buff + 18) = a5;              /* パレット番号 */
250:         }
251:     } else {                                /* 開始点,終了点指示なし */
252:         if (*(buff + 9) & 0x40){            /* タイル・パターン指示あり */
253:             *(buff + 18) = a1;              /* タイル・パターン長 */
254:             *(int *)(buff + 19) = a2;       /* 格納域オフセット */
255:             *(int *)(buff + 21) = a3;       /* 格納域セグメント */
256:         } else if (*(buff + 9) & 0x20) {    /* 塗りつぶし指示 */
257:             *(buff + 18) = a1;              /* パレット番号 */
258:         }
259:     }
260:     sys_call (0xA8, buff);
261: }
262:
263: /***********************************************************************
264: *
265: *    関数名：  paint1 ()
266: *    機  能：  塗りつぶしを色で行う
267: *    入  力：  int p1(x)     ･･･ 領域X座標
268: *              int p2(y)     ･･･ 領域Y座標
269: *              int p3(pal1)  ･･･ 領域色
270: *              int p4(pal2)  ･･･ 境界色
271: *    出  力：  なし
272: *
273: ************************************************************************/
274: void paint1 ()
275: {
276:     char buff [10];
277:     char work [WORK_SIZE];
278:
279:     *(int *)(buff + 0) = get_val ();              /* 開始点X座標 */
280:     *(int *)(buff + 2) = get_val ();              /* 開始点Y座標 */
281:     *(buff + 4) = get_val ();                     /* 領域色パレット番号 */
282:     *(buff + 5) = get_val ();                     /* 境界色パレット番号 */
283:     /* 最終オフセット */
284:     *(int *)(buff + 6) = (int)(work + WORK_SIZE - DS_OFF);
285:     /* 先頭オフセット */
286:     *(int *)(buff + 8) = (int)(work - DS_OFF);
287:     sys_call (0xA9, buff);
288: }
289:
290: /***********************************************************************
291: *
292: *    関数名：  paint2 ()
293: *    機  能：  タイルパターンでの塗りつぶし
294: *    入  力：  int p1(x)     ･･･ 開始X座標
295: *              int p2(y)     ･･･ 開始Y座標
```

〔リストB-7〕プログラム glio.c ⑥

```
296: *          int p3(len)  ･･･ パターン長
297: *          int p4(pal)  ･･･ 境界色
298: *          int p5(off)  ･･･ 格納オフセット
299: *          int p6(seg)  ･･･ 格納セグメント
300: *    出  力：  なし
301: *
302: ***********************************************************************/
303: void paint2 ()
304: {
305:     char buff [20];
306:     char work [WORK_SIZE];
307:
308:     *(int *)(buff + 0) = get_val ();         /* 開始点X座標 */
309:     *(int *)(buff + 2) = get_val ();         /* 開始点Y座標 */
310:     *(buff + 5) = get_val ();                /* タイルパターン長さ */
311:     *(int *)(buff + 6) = get_val ();         /* 格納域セグメント */
312:     *(int *)(buff + 8) = get_val ();         /* 格納域オフセット */
313:     *(buff + 10) = get_val ();               /* 境界色パレット番号 */
314:     /* 最終オフセット */
315:     *(int *)(buff + 6)= (int)(work + WORK_SIZE - DS_OFF);
316:     /* 先頭オフセット */
317:     *(int *)(buff + 8) = (int)(work - DS_OFF);
318:     sys_call (0xAB, buff);
319: }
320:
321: /**********************************************************************
322: *
323: *    関数名：   get ()
324: *    機  能：   描画情報の格納
325: *    入  力     int p1(x1)  ･･･ 左上X座標
326: *               int p2(y1)  ･･･ 左上Y座標
327: *               int p3(x2)  ･･･ 右下X座標
328: *               int p4(y2)  ･･･ 右下Y座標
329: *               int p5(off) ･･･ バッファのオフセット
330: *               int p6(seg) ･･･ バッファのセグメント
331: *               int p5(len) ･･･ 長さ
332: *    出  力：   なし
333: *
334: ***********************************************************************/
335: void get ()
336: {
337:     char buff [14];
338:
339:     *(int *)(buff +0) = get_val ();           /* 左上点X座標 */
340:     *(int *)(buff + 2) = get_val ();          /* 左上点Y座標 */
341:     *(int *)(buff + 4) = get_val ();          /* 右下点X座標 */
342:     *(int *)(buff + 6) = get_val ();          /* 右下点Y座標 */
343:     *(int *)(buff + 8) = get_val ();          /* 格納域オフセット */
344:     *(int *)(buff + 10) = get_val ();         /* 格納域セグメント */
345:     *(int *)(buff + 12) = get_val ();         /* 格納域長さ */
346:     sys_call (0xAB, buff);
347: }
348:
349: /**********************************************************************
350: *
351: *    関数名：   put1 ()
352: *    機  能：   描画情報の表示
353: *    入  力：   int p1(x)      ･･･ 左上X座標
354: *               int p2(y)      ･･･ 左上Y座標
355: *               int p2(mode)   ･･･ モード
356: *               int p3(sw)     ･･･ スイッチ
357: *               int p2(col1)   ･･･ フォアグランド色
358: *               int p3(col2)   ･･･ バックグランド色
359: *               int p5(off)    ･･･ バッファのオフセット
360: *               int p6(seg)    ･･･ バッファのセグメント
361: *               int p5(len)    ･･･ 長さ
362: *    出  力：   なし
363: *
364: ***********************************************************************/
365: void put1 ()
366: {
367:     char buff [14];
368:
369:     *(int *)(buff + 0) = get_val ();          /* 左上点X座標 */
370:     *(int *)(buff + 2) = get_val ();          /* 左上点Y座標 */
371:     *(int *)(buff + 4) = get_val ();          /* 格納域オフセット */
```

〔リストB-7〕プログラム glio.c ⑦

```
372:      *(int *)(buff + 6) = get_val ();           /* 格納域セグメント */
373:      *(int *)(buff + 8) = get_val ();           /* 格納域長さ */
374:      *(buff + 10) = get_val ();                 /* 描画モード */
375:      *(buff + 11) = get_val ();                 /* カラー・スイッチ */
376:      if (*(buff + 11) == 1){                    /* カラー指定あり */
377:           *(buff + 12) = get_val ();            /* フォア・グランド色 */
378:           *(buff + 13) = get_val ();            /* バックグランド色 */
379:      }
380:      sys_call (0xAC, buff);
381: }
382: /*****************************************************************
383:  *
384:  *   関数名： put2 ()
385:  *   機  能： 日本語の表示
386:  *   入  力： int p1(x)     ・・・ 左上X座標
387:  *            int p2(y)     ・・・ 左上Y座標
388:  *            int p2(mode)  ・・・ モード
389:  *            int p3(sw)    ・・・ スイッチ
390:  *            int p2(col1)  ・・・ フォアグランド色
391:  *            int p3(col2)  ・・・ バックグランド色
392:  *            int p2(code)  ・・・ JISコード
393:  *   出  力： なし
394:  *
395:  *****************************************************************/
396: void put2 ()
397: {
398:      char buff [10];
399:
400:      *(int *)(buff + 0) = get_val ();           /* 左上点X座標 */
401:      *(int *)(buff + 2) = get_val ();           /* 左上点Y座標 */
402:      *(int *)(buff + 4) = get_val ();           /* 日本語JISコード */
403:      *(buff + 6) = get_val ();                  /* 描画モード */
404:      *(buff + 7) = get_val ();                  /* カラー・スイッチ */
405:      if (*(buff + 7) == 1){                     /* カラー指定あり */
406:           *(buff + 8) = get_val ();             /* フォア・グランド色 */
407:           *(buff + 9) = get_val ();             /* バックグランド色 */
408:      }
409:      sys_call (0xAD, buff);
410: }
411: /*****************************************************************
412:  *
413:  *   関数名： roll ()
414:  *   機  能： 描画画面の移動
415:  *   入  力： int p1(v)     ・・・ 縦
416:  *            int p2(h)     ・・・ 横
417:  *            int p3(flg)   ・・・ フラグ
418:  *   出  力： なし
419:  *
420:  *****************************************************************/
421: void roll ()
422: {
423:      char buff[5];
424:
425:      *(int *)(buff + 0) = get_val ();           /* 上下ドット数 */
426:      *(int *)(buff + 2) = get_val ();           /* 左右ドット数 */
427:      *(buff + 4) = get_val ();                  /* フラグ*/
428:      sys_call (0xAE, buff);
429: }
430:
431: /*****************************************************************
432:  *
433:  *   関数名： point2 ()
434:  *   機  能： ドットに対するパレット番号の通知
435:  *   入  力： int p1(x) ・・・ ドットのX座標
436:  *            int p2(y) ・・・ ドットのY座標
437:  *   出  力： なし
438:  *
439:  *****************************************************************/
440: void point2 ()
441: {
442:      char buff [4];
443:
444:      *(int *)(buff + 0) = get_val ();          /* ドットX座標 */
445:      *(int *)(buff + 2) = get_val ();          /* ドットY座標 */
446:      sys_call (0xAF, buff);
447: }
```

● graphsub.asm モジュール

リストB-8のモジュールは，GCドライバのアセンブリ言語サブルーチンで，主としてCPUレジスタの設定や読み出し，およびシステム・コールの呼び出しを行っています．

CPUレジスタの操作やシステム・コールは，C言語のみによっても記述可能です．しかし，レジスタ操作に関しては，アセンブリ言語のほうがレジスタとパラメータの対応が取りやすいため，ここではあえてアセンブリ言語で記述しました．

同リストにおいて，ストラクチャDEV_HEADはデバイス・ヘッダのデータ構造を定義していて，オリジナルCONデバイス・ドライバを呼び出す際に使用しています．

データ・セグメントには，オリジナルCONデバイス・ドライバのストラテジ・ルーチンと割り込みエントリ・ルーチンのアドレスを格納するためのバッファが確保されます．

サブルーチン(関数)con_callは，引数として渡されたオリジナルCONデバイス・ドライバのアドレスとコマンド・パケットを用いて，MS-DOSになったつもりでオリジナルCONデバイス・ドライバを呼び出しています．このサブルーチンは，デバイス・ドライバに対するREADコマンドやNON-DESTRUCTIVE READコマンドを処理する際にコールされます．

サブルーチン(関数)get_offは，グラフLIOのテーブルから実際の処理アドレス(オフセット)を読み出し，そのアドレスをAXレジスタに返します．

サブルーチン(関数)set_vectは，ファンクション・リクエスト25Hを用いて，割り込みベクタの設定を行います．

〔リストB-8〕プログラム graphsub.asm ①

```
 1: ;*************************************************************
 2: ;
 3: ;    機    能 :   グラフィック・ドライバのサブルーチン
 4: ;    生    成 :   masm /ML graphsub;
 5: ;
 6: ;*************************************************************
 7:               .MODEL  SMALL, C
 8: ;*************************************************************
 9: ;
10: ;    構造体 :    DEV_HEAD
11: ;    機  能 :    デバイス・ヘッダの構造定義
12: ;
13: ;*************************************************************/
14: DEV_HEAD      STRUC
15: dev_link      DD        ?
16: dev_atr       DW        ?
17: stra_ptr      DW        ?
18: entry_ptr     DW        ?
19: DEV_HEAD      ENDS
20: ;*************************************************************
21: ;
22: ;    SEGMENT:   データ・セグメント
23: ;    機  能 :   オリジナルCONデバイスのアドレス
24: ;
25: ;*************************************************************/
26:               .DATA
27: org_strtgy    DD        ?
28: org_entry     DD        ?
29:
30: ;*************************************************************
31: ;
32: ;    SEGMENT:   コード・セグメント
33: ;    機  能 :   サブルーチン群
34: ;
35: ;*************************************************************/
36:               .CODE
37: ;*************************************************************
38: ;
39: ;    ルーチン名 :   con_call
40: ;    機  能 :      オリジナルCONデバイスの呼び出し
41: ;    入  力 :      arg1 … オリジナルCONデバイスのヘッダ
42: ;                  arg2 … コマンド・パケット
43: ;    出  力 :      なし
44: ;
45: ;*************************************************************
46: con_call      PROC      arg1:FAR PTR, arg2:FAR PTR
47:               les       bx, arg1
48:               mov       ax, es:[bx.stra_ptr]
49:               mov       WORD PTR org_strtgy, ax
50:               mov       WORD PTR org_strtgy + 2, es
```

〔リストB-8〕プログラム graphsub.asm ②

```
51:              mov        ax, es:[bx.entry_ptr]
52:              mov        WORD PTR org_entry, ax
53:              mov        WORD PTR org_entry + 2, es
54:              les        bx, arg2
55:              call       org_strtgy
56:              call       org_entry
57:              ret
58: con_call     ENDP
59:
60: ;*************************************************************
61: ;
62: ;    ルーチン名 :   get_off
63: ;    機    能 :   グラフＬＩＯのオフセット取得
64: ;    入    力 :   arg1 ・・・ＬＩＯテーブル・オフセット
65: ;                 arg2 ・・・ＬＩＯテーブル・セグメント
66: ;    出    力 :   AX   ・・・ＬＩＯオフセット
67: ;
68: ;*************************************************************
69: get_off      PROC       USES ES DI, arg1:WORD, arg2:WORD
70:              mov        di, arg1
71:              mov        es, arg2
72:              mov        ax, es:[di]
73:              ret
74: get_off      ENDP
75:
76: ;*************************************************************
77: ;
78: ;    ルーチン名 :   set_vect
79: ;    機    能 :   割り込みベクタの設定
80: ;    入    力 :   arg1 ・・・割り込みベクタ番号
81: ;                 arg2 ・・・割り込み処理ルーチンのオフセット
82: ;                 arg3 ・・・割り込み処理ルーチンのセグメント
83: ;    出    力 :   なし
84: ;
85: ;*************************************************************
86: set_vect     PROC       USES ds, arg1:WORD, arg2:WORD, arg3:WORD
87:              mov        ax, arg1
88:              mov        ds, arg2
89:              mov        dx, arg3
90:              mov        ah, 25H
91:              int        21h                     ;割り込みベクタの設定
92:              ret
93: set_vect     ENDP
94:
95: ;*************************************************************
96: ;
97: ;    ルーチン名 :   sys_call
98: ;    機    能 :   グラフＬＩＯの機能コール
99: ;    入    力 :   arg1 ・・・ベクタ番号
100: ;                arg2 ・・・パラメータへのポインタ
101: ;   出    力 :   AX   ・・・終了ステータス（00H: 正常終了）
102: ;
103: ;*************************************************************
104: sys_call     PROC       USES DS ES DI SI, arg1:WORD, arg2:PTR
105:              mov        ax, arg1
106:              mov        BYTE PTR cs:int_xx + 1, al  ;割り込み番号設定
107:              mov        bx, arg2
108:              mov        ax, ds
109:              add        ax, 2                   ;20Hバイト増加
110:              mov        ds, ax
111:              sub        bx, 20h                 ;20Hバイト減少
112:              push       bp
113: int_xx:
114:              int        00h                     ;00Hはダミー
115:              pop        bp
116:              mov        al, ah
117:              xor        ah, ah
118:              ret
119: sys_call     ENDP
120:
121: ;*************************************************************
122: ;
123: ;    ルーチン名 :   set_c5
124: ;    機    能 :   INT C5H のベクタ設定
125: ;    入    力 :   なし
126: ;    出    力 :   なし
127: ;
128: ;*************************************************************
129: set_c5       PROC       USES DS
```

〔リストB-8〕プログラム graphsub.asm ③

```
130:                push    cs
131:                pop     ds
132:                lea     dx, int_c5
133:                mov     al, 0C5h
134:                mov     ah, 25h
135:                int     21h                     ;割り込みベクタの設定
136:                ret
137: set_c5         ENDP
138:
139: ;*************************************************************
140: ;
141: ;    ルーチン名 :  int_c5
142: ;    機　能 :  INT C5H 処理ルーチン
143: ;    入　力 :  なし
144: ;    出　力 :  なし
145: ;
146: ;*************************************************************
147: int_c5         PROC    FAR
148:                iret
149: int_c5         ENDP
150:
151: ;*************************************************************
152: ;
153: ;    ルーチン名 :  chk_read
154: ;    機　能 :  キーボード・ステータスのチェック
155: ;    入　力 :  なし
156: ;    出　力 :  AX ・・・ 00H:  データなし
157: ;              AX ・・・ 01H:  データあり
158: ;
159: ;*************************************************************
160: chk_read       PROC
161:                mov     ah, 01h
162:                int     18h                     ;バッファ状態のセンス
163:                mov     al, bh
164:                xor     ah, ah
165:                ret
166: chk_read       ENDP
167:
168: ;*************************************************************
169: ;
170: ;    ルーチン名 :  sdsp
171: ;    機　能 :  文字列の表示
172: ;    入　力 :  arg1  文字列へのポインタ
173: ;    出　力 :  なし
174: ;
175: ;*************************************************************
176: sdsp           PROC    arg1:PTR
177:                mov     bx, arg1                ;ポインタ
178: sdsp_loop:
179:                mov     al, [bx]
180:                or      al, al                  ;'¥0'(文字列の終端) ?
181:                je      sdsp_exit
182:                cmp     al, 0Ah                 ;LFコード?
183:                jne     sdsp_next
184:                mov     al, 0Dh                 ;CRコード
185:                int     29h
186:                mov     al, 0Ah
187: sdsp_next:
188:                int     29h
189:                inc     bx
190:                jmp     sdsp_loop
191: sdsp_exit:
192:                ret
193: sdsp           ENDP
194:
195: ;*************************************************************
196: ;
197: ;    ルーチン名 :  cdsp
198: ;    機　能 :  1文字の表示
199: ;    入　力 :  arg1 ・・・ 文字コード
200: ;    出　力 :  なし
201: ;
202: ;*************************************************************
203: cdsp           PROC    arg1:WORD
204:                mov     ax, arg1                ;文字コード
205:                int     29h                     ;文字表示
206:                ret
207: cdsp           ENDP
208:                END
```

サブルーチン(関数)sys_callは，グラフLIOの機能を呼び出すためにソフトウェア割り込みを発行します．前述したように，ここではデバイス・ヘッダを保護するために，DSレジスタ値を20Hバイト分だけオフセットさせ，これにともなってパラメータへのポインタであるBXレジスタ値も20Hバイト分だけオフセットしています．

ここで，引数として与えられた割り込みベクタ番号に対応した割り込みを発生させる方法として，スタックを操作してIRET命令やFAR RET命令を実行する方法も考えられます．しかし，ここではプログラムの読みやすさに重点をおき，命令コードのオペランドを書き換える方法によって，指定されたソフトウェア割り込みの発行を実現しています．

サブルーチン(関数)set_c5は，ファンクション・リクエスト25Hを用いてINT C5Hベクタの設定を行います．ここで，INT C5H処理ルーチンは，プロシージャint c5なので，そのアドレスを設定しています．

プロシージャint c5はINT C5Hの処理ルーチンです．GCドライバでは，このINT C5Hで何もすることがないので，単にIRET命令を実行しているだけです．

サブルーチン(関数)chk_readは，キーボード・バッファの状態を調べます．ここでは，キーボードの状態を調べるのにBIOSのINT 18Hを用いて処理しています．INT 18Hでは，キーボード・バッファの状態がBHレジスタに返され，その内容が01Hの場合にはバッファにキー・データがあることを示し，00Hの場合にはバッファが空の状態であることを表しています．ここでは，BHレジスタの内容をAXレジスタに設定して戻り値として返しています．

サブルーチン(関数)sdspは，引数で渡されたポインタの指す文字列をスクリーンに表示します．ここでは，文字の表示にファンクション・リクエスト02Hや09Hなどは使用できない(GCドライバが再帰的に呼び出される)ため，システム・コールのINT 29Hを用いています．また，ここで扱う文字列はASCIZ文字列である必要があります．

サブルーチン(関数)cdspは，システム・コールINT 29Hを用いて1文字の表示を行います．

● st.asmモジュール

リストB-9は，GCドライバをデバッグするためのメイン・モジュールの記述例を示しています．このモジュールでは，**リストB-1**(graph.mak)に示したMAKEファイルによって自動的にアセンブル/コンパイルおよびリンク処理が行われ，結果としてstgraph.exeが実行モジュールとして生成されます．これによって，デバイス・ドライバの処理部分をCodeViewを用いてデバッグすることが可能となります．

このモジュールでは，MS-DOSになったつもりでデバイス・ドライバを呼び出していかなければなりません．したがって，リクエスト・ヘッダを含むコマンド・パケットを作成し，ストラテジ・ルーチンの呼び出し，割り込みエントリ・ルーチンの呼び出しを行います．

また，このモジュールにおけるコード・セグメントは，**リストB-2**のセグメント名あるいはクラス名とは別名にしておく必要があります．もし，同名のセグメント名やクラス名を指定すると，ドライバ本体のコード・セグメントと一緒に扱われ，デバイス・ヘッダの

〔リストB-9〕プログラムst.asm ①

```
 1: ;*****************************************************************
 2: ;
 3: ;    機     能 :  グラフィック・ドライバのデバッグ用
 4: ;    サ     ブ :  graph.asm graphsub.asm graphcc.c
 5: ;    生     成 :  make graphdbg.mak
 6: ;    使用方法 :  cv graph
 7: ;
 8: ;*****************************************************************/
 9:                PAGE    60, 130
10: ;*****************************************************************
11: ;
12: ;     構造体 :   REQ_HEAD
13: ;     機   能 :   リクエスト・ヘッダの定義
14: ;
15: ;*****************************************************************/
16: REQ_HEAD       STRUC                          ;リクエスト・ヘッダ
17: pac_len        DB        ?                    ;パケット長
18: dev_code       DB        ?                    ;デバイス・コード
19: com_code       DB        ?                    ;コマンド・コード
20: status         DW        ?                    ;ステータス
21: reserve        DB        8 dup (?)            ;リザーブ
22: REQ_HEAD       ENDS
23:
```

〔リストB-9〕プログラム st.asm ②

```
 24: ;*****************************************************************
 25: ;
 26: ;     構造体：  INIT_PACKET
 27: ;     機  能：  INIT コマンド用パケットの定義
 28: ;
 29: ;****************************************************************/
 30: INIT_PACKET STRUC                                  ;INIT コマンド用パケット
 31: init_head   DB      SIZE REQ_HEAD DUP (?)          ;リクエスト・ヘッダ
 32: unit        DB      1                              ;ユニット数
 33: dev_end     DD      ?                              ;エンド・アドレス
 34: bpb         DD      ?                              ;ＢＰＢ配列へのポインタ
 35: dev_num     DB      ?                              ;ドライブ番号
 36: INIT_PACKET ENDS
 37:
 38: ;*****************************************************************
 39: ;
 40: ;     構造体：  RW_PACKET
 41: ;     機  能：  READ / WRITE コマンド用パケットの定義
 42: ;
 43: ;****************************************************************/
 44: RW_PACKET   STRUC                                  ;BUILD BPB 用パケット
 45: rw_head     DB      SIZE REQ_HEAD DUP (?)          ;リクエスト・ヘッダ
 46: rw_disc     DB      1                              ;メディア・デスクリプタ
 47: rw_trans    DD      ?                              ;転送アドレス
 48: bytes       DW      ?                              ;転送バイト
 49: sct_bgn     DW      ?                              ;開始セクタ（無視）
 50: rw_id       DD      ?                              ;ＩＤへのポインタ
 51: RW_PACKET   ENDS
 52:
 53: ;*****************************************************************
 54: ;
 55: ;     SEGMENT:  データ・セグメント
 56: ;     機  能：  コマンド・パケットの確保
 57: ;
 58: ;****************************************************************/
 59: _DATA1          SEGMENT WORD PUBLIC 'DATA1'
 60: config          DB      '¥wk1¥emm¥ram.sys', 0, '200', 0Dh, 0Ah, 00h
 61: init_com        INIT_PACKET     <>
 62: wr_com          RW_PACKET       <>
 63: str0            DB      1Bh, 07, 'circle 200 200 180 90 1 32 2', 0Dh, 0Ah
 64: str1            DB      1Bh, 07, 'color2 2 1', 0Dh, 0Ah
 65: str2            DB      1Bh, 07, 'line 100 100 600 300 5 2 0 4', 0Dh, 0Ah
 66: str3            DB      1Bh, 07, 'paint1 410 10 2 1', 0Dh, 0Ah
 67: str4            DB      1Bh, 07,'cls' 0Dh, 0Ah
 68: buff            DB      256 DUP (?)
 69: _DATA1          ENDS
 70:
 71: ;*****************************************************************
 72: ;
 73: ;     SEGMENT:  スタック・セグメント
 74: ;     機  能：  スタックの確保
 75: ;
 76: ;****************************************************************/
 77: _STACK          SEGMENT WORD STACK 'STACK'
 78: _stack          DW      256 DUP (?)
 79: _STACK          ENDS
 80:
 81: ;*****************************************************************
 82: ;
 83: ;     SEGMENT:  コード・セグメント
 84: ;     機  能：  ドライバの呼び出し
 85: ;
 86: ;****************************************************************/
 87:                 EXTRN   _strategy:FAR, _entry:FAR
 88:                 EXTRN   _dev_header:DWORD
 89:                 PUBLIC  _begin
 90:
 91: _TEXT1          SEGMENT WORD PUBLIC 'TEXT1'
 92:                 ASSUME  CS:_TEXT1, DS:_DATA1, ES:_DATA1, SS:_STACK
 93:
 94: _begin          PROC
 95:                 mov     cs:WORD PTR _dev_header, 0      ;最初のリンク
 96:                 mov     cs:WORD PTR _dev_header + 2, 12A1h
 97:                 ;
 98:                 ; INIT コマンドの呼び出し
 99:                 ;
100:                 mov     ax, SEG init_com            ;コマンド・パケットのセグメント
101:                 mov     es, ax
102:                 lea     bx, init_com                ;コマンド・パケットのオフセット
```

〔リストB-9〕プログラム st.asm ③

```
103:            call     _strategy
104:            mov      es:[bx.pac_len], SIZE INIT_PACKET
105:            mov      es:[bx.com_code], 0
106:            call     _entry              ;INIT コマンド実行
107:
108:            ;
109:            ; WRITE コマンドの呼び出し
110:            ; GCIRCLE
111:            ; 楕円の描画
112:            ;
113:            mov      es:[bx.pac_len], SIZE RW_PACKET
114:            mov      es:[bx.com_code], 8
115:            mov      es:[bx.bytes], 1              ;バイト数
116:            mov      WORD PTR es:[bx.rw_trans + 2], SEG str0
117:            mov      cx, 32                        ;バイト数
118:            mov      di, OFFSET str0               ;楕円の描画
119: loop0:
120:            mov      WORD PTR es:[bx.rw_trans], di
121:            call     _entry                        ;WRITE コマンド実行
122:            inc      di
123:            loop     loop0
124:
125:            ;
126:            ; GCOLOR2
127:            ; パレット番号の指定
128:            ;
129:            mov      WORD PTR es:[bx.rw_trans + 2], SEG str1
130:            mov      cx, 14                        ;バイト数
131:            mov      di, OFFSET str1               ;COLOR2の実行
132: loop1:
133:            mov      WORD PTR es:[bx.rw_trans], di
134:            call     _entry                        ;WRITE コマンド実行
135:            inc      di
136:            loop     loop1
137:
138:            ;
139:            ; GLINE
140:            ; 箱の塗りつぶし
141:            ;
142:            mov      WORD PTR es:[bx.rw_trans + 2], SEG str2
143:            mov      cx, 32                        ;バイト数
144:            mov      di, OFFSET str2
145: loop2:
146:            mov      WORD PTR es:[bx.rw_trans], di
147:            call     _entry                        ;WRITE コマンド実行
148:            inc      di
149:            loop     loop2
150:
151:            ;
152:            ; GPAINT1
153:            ; 塗りつぶし
154:            ;
155:            mov      WORD PTR es:[bx.rw_trans + 2], SEG str3
156:            mov      cx, 21                        ;バイト数
157:            mov      di, OFFSET str3
158: loop3:
159:            mov      WORD PTR es:[bx.rw_trans], di
160:            call     _entry                        ;WRITE コマンド実行
161:            inc      di
162:            loop     loop3
163:
164:            ;
165:            ; GCLS
166:            ; グラフィック画面の消去
167:            ;
168:            mov      WORD PTR es:[bx.rw_trans + 2], SEG str4
169:            mov      cx, 7                         ;バイト数
170:            mov      di, OFFSET str4
171: loop4:
172:            mov      WORD PTR es:[bx.rw_trans], di
173:            call     _entry                        ;WRITE コマンド実行
174:            inc      di
175:            loop     loop4
176:
177:            mov      ax, 4C00h                     ;プログラム終了
178:            int      21h
179: _begin     ENDP
180: _TEXT1     ENDS
181:            END      _begin
```

配置が意図したとおりになりません．

GC ドライバでは，オリジナルの CON デバイス・ドライバの検索を行っているために，直前のデバイス・ドライバへのポインタを GC ドライバ自身のデバイス・ヘッダに設定してやらなければなりません．

この直前のデバイス・ドライバのアドレスは，GC ドライバの INIT コマンド処理ルーチン(関数 init)に自身のアドレス(CS レジスタ値)を表示するようにプログラミングし，デバイス登録することによって知ることができます．

GC ドライバ自身の組み込みアドレスが得られたら，CodeView や SYMDEB などのデバッガを用いて，そのリンク情報をたどれば，すべてのデバイス・ドライバの実アドレスを知ることができます．

● gtestc.c モジュール

リストB-10 は，GC ドライバのアクセスを行うプログラムの例です．

このプログラムでは，グラフィック画面の初期化を行ったあと，7個の箱を描いてその色をペインティングにより変化させます．

次に7個の同心楕円を色を変えながら描き，描き終わった時点でパレット番号を書き換えることにより，あたかも画面がフラッシュしているかのごとく表現しています．そして，フィナーレでは画面を左にスクロールして終了します．この横スクロールの機能は BASIC においてもサポートされていないユニークな機能です．

これに BASIC の LOCATE 文にあたる関数を，MS-DOS 標準のエスケープ・シーケンスを利用して作成すれば，BASIC 感覚のグラフィックスが高速に表現できるので，処理結果を視覚的に把握したいようなアプリケーションには重宝するものと思います．

同リストにおいて，関数 main はテスト・プログラムの統括制御を行います．

まず，最初に MS-DOS 標準のエスケープ・シーケンスによりテキスト画面の消去を行います．次に，グラフ LIO の CLS コマンドを用いてグラフィック画面の消去を行います．

同リストでは，グラフィック制御文字列を表すエスケープ文字列をグローバル・ポインタ Gesc によって指すことにします．つづいて，グラフ LIO の G-

〔リストB-10〕プログラム gtest.c ①

```
  1: /**********************************************************************
  2: *
  3: *    機    能 :  グラフィック・ドライバのテスト・プログラム
  4: *    生    成 :  cl -AS gtest.c
  5: *
  6: ***********************************************************************/
  7: #define         COL_MAX 8
  8: #include        <stdio.h>
  9: #include        <stdlib.h>
 10:
 11: /********************************************************************
 12: *
 13: *       機    能:   グローバル変数の宣言
 14: *
 15: *********************************************************************/
 16: static char *Gesc = "¥033¥007";
 17:
 18: /********************************************************************
 19: *
 20: *       機    能:    ANSIプロトタイプ関数宣言
 21: *
 22: *********************************************************************/
 23: void main (void);
 24: void box (void);
 25: void scircle (void);
 26: void roll_l (void);
 27: void itaval (int);
 28:
 29: /********************************************************************
 30: *
 31: *       関数名:    main ()
 32: *       機  能:    テスト・プログラムの統括制御
 33: *       入  力:    なし
 34: *       出  力:    なし
 35: *
 36: *********************************************************************/
 37: void main ()
 38: {
 39:     puts( "¥033*¥033=  " ); /* テキスト画面の消去    */
 40:     printf ("%s cls¥n", Gesc);
 41:     printf ("%s screen 3 0 0 1¥n", Gesc);
```

〔リストB-10〕プログラム gtest.c ②

```
 42:     box();                      /* 箱   */
 43:     scircle ();                 /* 楕円   */
 44:     roll_l();                   /* 画面の左移動 */
 45:     printf ("%s cls¥n", Gesc);
 46:     exit (0);
 47: }
 48:
 49: /**************************************************************
 50:  *
 51:  *    関数名：   box ()
 52:  *    機  能：   箱の描画と塗りつぶし
 53:  *    入  力：   なし
 54:  *    出  力：   なし
 55:  *
 56:  **************************************************************/
 57: void box()
 58: {
 59:     int i, j;
 60:     int y1 = 0, y2 = 0;
 61:
 62:     for (i = 1; i < COL_MAX; i++){
 63:         y2 = y1 + 50;
 64:         /* 箱を描く   */
 65:         printf ("%s line 400 %d 600 %d %d 1 0¥n", Gesc, y1, y2, i);
 66:         y1 = y1 + 55;
 67:     }
 68:     for (j = 1; j < COL_MAX; j++){
 69:         y1 = 0;
 70:         for (i = 1; i < COL_MAX; i++){
 71:             /* 箱を塗りつぶす */
 72:             printf ("%s paint1 410 %d %d %d¥n", Gesc, y1 + 10, i + j, i);
 73:             y1 = y1 + 55;
 74:         }
 75:     }
 76: }
 77:
 78: /**************************************************************
 79:  *
 80:  *    関数名：   scircle ()
 81:  *    機  能：   楕円の描画と塗りつぶし
 82:  *    入  力：   なし
 83:  *    出  力：   なし
 84:  *
 85:  **************************************************************/
 86: void scircle()
 87: {
 88:     int i, j, col2;
 89:
 90:     for(i = 1; i < COL_MAX; i++){
 91:         printf ("%s circle 200 200 %d %d %d %d %d¥n",
 92:             Gesc, 200 - i * 20, 100 - i * 10, i, 0x20, i + 1);
 93:     }
 94:     for (j = 1; j < 20; j++) {
 95:         for (i = 1; i < COL_MAX; i++) {
 96:             col2 = rand () % 7 + 1;
 97:             printf ("%s color2 %d %d¥n", Gesc, i, col2);  /* パレット番号の指定 */
 98:             printf ("%s color2 %d %d¥n", Gesc, col2, i);  /* もとの色に */
 99:         }
100:     }
101: }
102:
103: /**************************************************************
104:  *
105:  *    関数名：   roll_l ()
106:  *    機  能：   画面の左へのスクロール
107:  *    入  力：   なし
108:  *    出  力：   なし
109:  *
110:  **************************************************************/
111: void roll_l ()
112: {
113:     int i;
114:
115:     for(i = 0; i < 17; i ++) {
116:         printf ("%s roll 0, 40, 0¥n", Gesc);
117:     }
118: }
```

SCREEN コマンドを実行して 640×400 ドットのカラー画面に設定します．

次に，関数 box を呼んで箱の描画を行い，関数 scircle を呼んで楕円の描画とパレット番号の書き換えを行います．そして，関数 roll_1 を呼んでグラフィック画面を左側にスクロールします．

すべての処理が終わったら，再びグラフ LIO の GCLS コマンドを実行して，グラフィック画面を消去してプログラムを終了します．

関数 box は，箱の描画と塗りつぶしを行います．まず，for ループとグラフ LIO の GLINE コマンドを用いて 7 個の箱を描画します．次に，for ループとグラフ LIO の GPAINT コマンドを用いてあらかじめ描かれた箱に対して，次々と色を塗り替えていきます．

関数 scircle は，7 個の同心楕円の描画とパレット番号の変更を行います．まず，for ループとグラフ LIO の GCIRCLE コマンドを用いて，色を変えながら同心楕円の描画を行います．次に，for ループとグラフ LIO の GCOLOR2 コマンドを用いてパレット番号の変更を行い，次々に色を変化させます．このパレット番号の変更で，グラフィック画面がフラッシュしているような効果を得ることができます．

関数 roll_1 はグラフィック画面を左にスクロールします．ここでは，for ループとグラフ LIO の GROLL コマンドを用いて，グラフィック画面を少しずつ左にスクロールさせています．

● MS-DOS 標準のエスケープ・シーケンス ●

MS-DOS では，ASCII 制御コードとは別に，表 D のエスケープ文字列をコンソールに送ることにより，画面の様々な制御を行うことが可能になっています．

〔表 D〕エスケープ・シーケンス

エスケープ文字列	機　能
ESC [pl ; pcH または ESC [pl ; pcf	カーソルを pl 行 pc カラムに移動する
ESC＝lc	カーソルを l 行 c カラムに移動．l と c は 2 進数であり 20H のオフセットを加える
ESC [pnA	カーソルを同一カラムで pn 行上に移動する
ESC [pnB	カーソルを同一カラムで pn 行下に移動する
ESC [pnC	カーソルを同一行で pn 分だけ右に移動する
ESC [pnD	カーソルを同一行上で pn 分だけ左に移動する
ESC [0J	カーソル位置から最終行の右端までクリアする
ESC [1J	先頭行の左端からカーソル位置までをクリアする
ESC [2J または ESC *	CTR 画面をすべてクリアし，カーソルをホーム位置に移動する
ESC [0K	カーソル位置から同一行の右端までをクリアする
ESC [1K	同一行の左端からカーソル位置までをクリアする
ESC [2K	カーソルのある行の左端から右端までをクリアする
ESC [pnM	カーソルのある行から pn 行分だけ下の行を削除し，それ以降の行を上に詰める
ESC [pnL	カーソルのある行を pn 行分だけ下に移動し，空白行を挿入する
ESC D	カーソルを同一カラムで 1 行だけ下に移動する
ESC E	カーソルを 1 行下の左端に移動する
ESC M	カーソルを同一カラムで 1 行だけ上に移動する
ESC [s	カーソル位置の情報(行，カラム，属性)を保存する
ESC [u	ESC [s で保存した内容を復帰する
ESC [6n	カーソル位置をコンソールで指定する．形式は ESC [pl ; pcR であり pl 行 pc カラムに移動する
ESC)0	漢字モードの設定
ESC)3	グラフ・モードの設定
ESC [>5l	カーソルを見えるように設定する(デフォルト)
ESC [>5h	カーソルが見えないモードに設定する
ESC [>1h	ファンクション・キーの表示行をユーザに開放する
ESC [>1l	ファンクション・キーの内容を表示する(デフォルト)
ESC [>3h	20 行表示モードに設定する
ESC [>3l	25 行表示モードに設定する
ESC [ps ;…; psm	表示文字の属性を設定する

ps の値：0＝既定の属性(デフォルト)，1＝ハイライト(モノクロのみ)，2＝バーチカル・ライン，4＝アンダ・ライン，5＝ブリンク，7＝リバース，30＝黒，31＝赤，32＝緑，33＝黄色，34＝青，35＝紫，36＝水色，37＝白(40〜47 はリバース)．

Appendix C RAMディスク・ドライバの作成

ここでは，拡張メモリの応用例としてRAMディスク・ドライバの実現方法について解説します．デバイス・ドライバの詳細については第8章を参照してください．

なお，ここで紹介するRAMディスク・ドライバは，config.sysファイルにDEVICEコマンドを用いて登録する際に，**図C-1**のようにオプションとしてディスク容量を指定できるようにプログラムされています．

RAMディスク・ドライバは，**図C-2**に示したように5個のソース・ファイルから構成されています．同図では，ドライバ本体のほかにドライバのデバッグ用プログラムの作成方法までを示しています．これらのプログラムの作成は，**リストC-1**のMAKEファイルとMAKEユーティリティによって自動的に行われます．

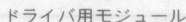
ここでも，RAMディスク・ドライバは，グラフィック・コンソール・ドライバと同様にC言語で記述してあり，C言語を用いてブロック型デバイス・ドライバを作成する際の参考になるようにしています．ここでもやはり，ドライバ構造に関する部分やCPUレジスタとのやりとりを行う部分はアセンブリ言語で記述し，制御構造を含むドライバの主要な部分はC言語を用いて記述しています．

〔図C-1〕ディスク容量の指定

config・sys

DEVICE=ram.sys ⌴ 1024

ドライバのファイル名（パス名でもよい）　ディスク容量（Kバイト）

〔図C-2〕RAMディスク・ドライバのモジュール構成

● RAMディスク・ドライバの構成

それぞれのソース・ファイルの機能は以下のとおりです．

① seg.h…セグメント配置の制御を行うためのヘッダ・ファイル．

② ram.asm…RAMディスク・ドライバのメイン・ルーチンともいえるプログラムで，デバイス・ヘッダやストラテジ・ルーチンおよび割り込みエントリ・ルーチンを含む．

③ struc.h…ramc.cファイルで取り込まれるヘッダ・ファイルで，BPBテーブルやリクエスト・ヘッダおよびコマンド・パケットなどのデータ構造が定義されている．

④ ramc.c…デバイス・コマンド・コードの解析と各コマンド・コードに対応した処理を行う．

⑤ ramsub.asm…アセンブリ言語サブルーチンで，主としてEMMファンクション・コールやCPUレジスタ値の読み出し/設定を行う．

⑥ st.asm…RAMディスク・ドライバをデバッグする際のスタートアップ・ルーチンとなり，コマンド・パケットを作成してデバイス・ドライバをアクセスするために，MS-DOSをエミュレーションする．

以下，これらの順にソース・リストを追いながら説明していきます．

● seg.h

リストC-2のヘッダ・ファイルは，セグメント配置の制御を行います．このセグメント配置の手順は，グラフィック・コンソール・ドライバとまったく同様です．

RAMディスク・ドライバは，アセンブリ言語とC言語で記述されていますが，デバイスドライバは64Kバイト以下でなければならず，またセグメントを参照するコードを含んでいてはなりません．

MS-Cでは，スモール・モデルの場合でも，コード・セグメントとデータ・セグメントは分けて扱っている

〔リストC-1〕
ドライバ作成用MAKEファイル

```
 1: st.obj: st.asm
 2:     masm /ZI/Z/ML st,, st;
 3:
 4: ram.obj: ram.asm seg.h
 5:     masm /ZI/Z/ML ram,, ram;
 6:
 7: ramsub.obj: ramsub.asm
 8:     masm /ZI/Z/ML ramsub,, ramsub;
 9:
10: ramc.obj: ramc.c struc.h
11     msc /Zi/Ze/Zl/Zp/Fa/Gs/DLINT_ARGS/J ramc.c;
12:
13: ram.exe: ram.obj ramc.obj ramsub.obj
14:     link /CO/NOI/MAP/LI ram ramc ramsub, ram, ram;
15:
16: stram.exe: st.obj ram.obj ramc.obj ramsub.obj
17:     link /CO/NOI/MAP/LI st ram ramc ramsub, stram, stram;
18:
19: ram.sys: ram.exe
20:     exe2bin ram.exe ram.sys
```

〔リストC-2〕
ヘッダ・ファイルseg.h

```
 1: ;*****************************************************************
 2: ;
 3: ;  機    能:  セグメント配置の制御
 4: ;  生    成:  masm /ML seg;
 5: ;
 6: ;****************************************************************/
 7:             PAGE     60, 130
 8: _TEXT       SEGMENT WORD PUBLIC 'CODE'
 9: _TEXT       ENDS
10: _DATA       SEGMENT WORD PUBLIC 'DATA'
11: _DATA       ENDS
12: CONST       SEGMENT WORD PUBLIC 'CONST'
13: CONST       ENDS
14: _BSS        SEGMENT WORD PUBLIC 'BSS'
15: _BSS        ENDS
16: c_common    SEGMENT WORD PUBLIC 'BSS'
17: c_common    ENDS
18: _dend       SEGMENT WORD PUBLIC 'BSS'
19: _dend       ENDS
20: DGROUP      GROUP      _TEXT, _DATA, CONST, _BSS, c_common, _dend
```

ため，実行時のデータ・セグメント(DSレジスタ値)を計算するのが少々面倒になります．

このため，RAMディスク・ドライバでは，ディレクティブGROUPを用いてすべてのセグメントをグループ化し，スタック・セグメントも合わせて64Kバイトの中に収めています．これによって，プログラム中で特にセグメント・レジスタを意識する必要がなく，8ビット・マシンにおけるプログラムと同様にリニア・アドレスとしてプログラミングできることになります．

また，セグメント_dendはグラフィック・コンソール・ドライバと同様に，RAMディスク・ドライバの終了アドレスを知るために定義されているセグメントで，リンクされたときにそのセグメントが最後に配置されるようにセグメント名を決めています(**リストC-3**参照)．

● **ram.asm**

リストC-4のram.asmモジュールは，RAMディスク・ドライバのメイン・モジュールであり，デバイス・ヘッダやストラテジ・ルーチンおよび割り込みエントリ・ルーチンが記述されています．

デバイス・ドライバの最終アドレスを示すラベルd_endは，セグメント_dend内で定義され，ramc.cモジュールから参照可能とするためPUBLIC宣言を行っておきます．また，ラベルentryやstrategyも，デバッグ用スタートアップ・ルーチン(st.asm)をリンクした際に参照可能とするため，PUBLIC宣言を行っておきます．

データ・セグメントには，ドライバで使用するスタック領域の確保を行います．デバイス・ドライバの場合，プログラムの先頭にはデバイス・ヘッダが位置しなければなりません．ここでは，コード・セグメント(デバイス・ヘッダ)よりも先にデータ・セグメントを定義していますが，**リストC-2**のヘッダ・ファイルによってあらかじめセグメント配置を指定しているために，リンク時にデータ・セグメントはコード・セグメントの後に配置されます．

コード・セグメントでは，まずデバイス・ヘッダの定義を行います．デバイス・ヘッダの先頭のダブル・ワードはデバイス・リンク用のポインタであり，通常このフィールドにはFFFFHを設定しておきます．

次のワード値は，デバイス属性を示すフィールドです．RAMディスク・ドライバはブロック・デバイスであり，また，IBMフォーマットとしているので，このフィールドには0000Hを設定します．

〔リストC-3〕MAPファイル

```
 LINK : warning L4021: no stack segment

  Start  Stop   Length Name                   Class
  00000H 00856H 00857H _TEXT                  CODE
  00858H 01227H 009D0H _DATA                  DATA
  01228H 01228H 00000H CONST                  CONST
  01228H 01228H 00000H _BSS                   BSS
  01228H 01235H 0000EH c_common               BSS
  01236H 01236H 00000H _dend                  BSS  ←(セグメントが最後に配置されることを確認する)

  Origin   Group
  0000:0   DGROUP

   Address         Publics by Name

  0000:056E        _atoi
  0000:029B        _bld_bpb
                .
                .

  0000:07F5        _write_sub
  0000:9876  Abs   __acrtused

   Address         Publics by Value

  0000:0015        _strategy
  0000:0020        _entry
                .
                .

  0000:1234        _Reg_dx
  0000:1236        _d_end ←(ラベルも最後に配置されることを確認する)
  0000:9876  Abs   __acrtused

 Line numbers for RAM.OBJ(ram.ASM) segment _TEXT
```

〔リストC-4〕プログラム ram.asm ①

```
 1: ;*************************************************************
 2: ;
 3: ;   機    能:  拡張メモリ対応RAMディスク・ドライバ(メイン)
 4: ;   サ    ブ:  seg.asm ramsub.asm ramc.c
 5: ;   生    成:  make ram.mak
 6: ;   使用方法:  DEVIVE = [ディスク容量(Kバイト)]
 7: ;
 8: ;*************************************************************/
 9:             PAGE      60, 130
10:             .MODEL    SMALL, C
11:             INCLUDE   seg.h
12:
13: STK_SIZE        EQU       2048
14: _acrtused       EQU       9876h
15:
16: ;*************************************************************
17: ;
18: ;   機    能:   外部参照
19: ;
20: ;*************************************************************/
21:             EXTRN     start:NEAR
22:             PUBLIC    d_end
23:             PUBLIC    _acrtused
24:             PUBLIC    entry, strategy
25:
26: ;*************************************************************
27: ;
28: ;   機    能:   ドライバの最終アドレスの定義
29: ;
30: ;*************************************************************/
31: _dend       SEGMENT WORD PUBLIC 'BSS'
32: d_end       LABEL     WORD
33: _dend       ENDS
34:
35: ;*************************************************************
36: ;
37: ;   SEGMENT:   データ・セグメント
38: ;   機    能:   スタックの確保
39: ;
40: ;*************************************************************/
41:             .DATA
42: stack       DB        STK_SIZE dup (?)
43: stk_btm     LABEL     WORD
44:
45: ;*************************************************************
46: ;
47: ;   SEGMENT:   コード・セグメント
48: ;   機    能:   デバイス ヘッダおよびプログラム
49: ;
50: ;*************************************************************/
51:             .CODE
52: ;*************************************************************
53: ;
54: ;   機    能:   デバイス・ヘッダ
55: ;
56: ;*************************************************************/
57: dev_header  LABEL     WORD
58:             DD        -1
59:             DW        0000h        ;ブロック・デバイス
60:             DW        strategy
61:             DW        entry
62:             DB        1            ;ユニット数
63:
64: ;*************************************************************
65: ;
66: ;   機    能:   バッファ
67: ;
68: ;*************************************************************/
69: data_seg    DW        ?
70: packet      DD        ?
71: sp_buff     DW        ?
72: ss_buff     DW        ?
73:
74: ;*************************************************************
75: ;
76: ;   ルーチン名:   starategy
77: ;   機    能:   ストラテジ・ルーチン
```

〔リストC-4〕プログラム ram.asm ②

```
 78: ;   入  力  ES:BX   コマンド・パケットへのポインタ
 79: ;   出  力  なし
 80: ;
 81: ;**************************************************************
 82: strategy    PROC    FAR
 83:             mov     WORD PTR cs:[packet], bx      ;コマンド・パケット格納
 84:             mov     WORD PTR cs:[packet + 2], es
 85:             ret
 86: strategy    ENDP
 87:
 88: ;**************************************************************
 89: ;
 90: ;   ルーチン名 :  entry
 91: ;   機   能 :  割り込みルーチン
 92: ;   入   力 :  なし
 93: ;   出   力 :  なし
 94: ;
 95: ;**************************************************************
 96: entry       PROC    FAR USES AX BX CX DX SI DI BP DS
 97:             push    es
 98:             cli
 99:             mov     cs:ss_buff, ss                ;スタック退避
100:             mov     cs:sp_buff, sp
101:             mov     ax, cs
102:             mov     ds, ax
103:             mov     es, ax
104:             mov     ss, ax
105:             lea     sp, stk_btm
106:             push    WORD PTR packet + 2
107:             push    WORD PTR packet
108:             call    start
109:             add     sp, 4
110:             mov     sp, cs:sp_buff
111:             mov     ss, cs:ss_buff
112:             sti
113:             pop     es
114:             ret
115: entry       ENDP
116:             END
```

次の二つのワード値は，それぞれストラテジ・ルーチンへのポインタと割り込みエントリ・ルーチンへのポインタが入ります．

RAMディスク・ドライバは，1ドライブのみのサポートとなっているため，デバイスのユニット数フィールドには“1”を設定しておきます．

デバイス・ヘッダの定義が終われば，そのほかのデータやプロシージャの配置には制限がありません．ここでは，デバイス・ヘッダの後にドライバが使用するワーク領域を確保しています．

プロシージャ strategy はストラテジ・ルーチンです．このプロシージャでは，ES：BX レジスタに渡されたコマンド・パケットへのポインタを RAM ディスク・ドライバのワーク・エリアに格納して FAR リターンしています．

プロシージャ entry は割り込みエントリ・ルーチンです．このプロシージャでは，SS レジスタと SP レジスタを RAM ディスク・ドライバのワーク・エリアに格納したのち，デバイス・ドライバのベース・セグメント値を SS，ES，DS の各セグメント・レジスタに設定します．

MS-DOS は，デバイス・ドライバを呼び出す際に，スタック領域を 20 個程度しか用意していないため，SP レジスタを RAM ディスク・ドライバのスタック領域に設定します．

次に，ramc.c モジュール内の関数 start を呼んでドライバに対するデバイス・コマンドの処理を行います．関数 start は C 言語で記述されているため，コマンド・パケットへの FAR ポインタをスタックに積んでから関数 start をコールします(スタックを介して関数 start への引数として渡す)．

コマンド・パケットの解析や処理は関数 start が行い，制御がプロシージャ entry に返されたら，SS：SP レジスタ値をもとの値に復元して MS-DOS に FAR リターンします．

● struc.h

リストC-5 のヘッダ・ファイルは，ramc.c モジュールで取り込まれて参照されるファイルで，BPB テーブルやリクエスト・ヘッダおよびコマンド・パケットのデータ構造を定義しています．

構造体 _BPB は，BPB テーブルのデータ構造を定

義しています．

構造体 _REQ_HEAD は，リクエスト・ヘッダのデータ構造を定義しています．

構造体 _INIT_PACKET は，INIT コマンド用のコマンド・パケットのデータ構造を定義しています．

構造体 _MDACHK は，MEDIA CHECK コマンド用のコマンド・パケットのデータ構造を定義していま す．

構造体 _BLDBPB は，BUILD BPB コマンド用のコマンド・パケットのデータ構造を定義しています．

構造体 _RW_PACKET は，READ コマンドおよび WRITE コマンド用のコマンド・パケットのデータ構造を定義しています．

〔リストC-5〕ヘッダ・ファイル struc.h ①

```
 1: /*****************************************************************
 2: *
 3: *   機   能 :  構造体の定義
 4: *
 5: *****************************************************************/
 6: /*****************************************************************
 7: *
 8: *      構 造 体 :   _BPB
 9: *      機   能 :   BPBテーブルの定義
10: *
11: *****************************************************************/
12: typedef struct _BPB {
13:     unsigned int  sector;         /* 1セクタあたりのバイト数 */
14:     unsigned char sec_clus;       /* 1クラスタあたりのセクタ数 */
15:     unsigned int  resv_sec;       /* 予備のセクタ数 */
16:     unsigned char fat_count;      /* FATの数 */
17:     unsigned int  dir_count;      /* ルート・ディレクトリのエントリ数 */
18:     unsigned int  cap;            /* 論理セクタ数 */
19:     unsigned char dev_disc;       /* メディア・ディスクリプタ */
20:     unsigned int  sec_fat;        /* 1FATあたりのセクタ数 */
21: } BPB;
22:
23: /*****************************************************************
24: *
25: *      構 造 体 :   _REQ_HEAD
26: *      機   能 :   リクエスト・ヘッダの定義
27: *
28: *****************************************************************/
29: typedef struct _REQ_HEAD {
30:     unsigned char packet_len;     /* コマンド・パケットの長さ */
31:     unsigned char dev_code;       /* 論理装置コード */
32:     unsigned char cmd_code;       /* コマンド・コード */
33:     unsigned int  status;         /* ステータス */
34:     unsigned char reserve[8];     /* 予約域 */
35: } REQ_HEAD;
36:
37: /*****************************************************************
38: *
39: *      構 造 体 :   _INIT_PACKET
40: *      機   能 :   INITコマンド用コマンド・パケットの定義
41: *
42: *****************************************************************/
43: typedef struct _INIT_PACKET {
44:     REQ_HEAD packet;              /* リクエスト・ヘッダ */
45:     unsigned char unit;           /* ユニット数 */
46:     char far *end;                /* エンド・アドレス */
47:     BPB  far *bpb;                /* BPB配列へのポインタ */
48:     unsigned char dev_num;        /* ブロック・デバイス番号 */
49: } INIT_PACKET;
50:
51: /*****************************************************************
52: *
53: *      構 造 体 :   _MDACHK_PACKET
54: *      機   能 :   MEDIA CHECKコマンド用コマンド・パケットの定義
55: *
56: *****************************************************************/
57: typedef struct _MDACHK_PACKET {
58:     REQ_HEAD packet;              /* リクエスト・ヘッダ */
59:     unsigned char media_dsc;      /* メディア・ディスクリプタ */
60:     unsigned char ret_code;       /* ドライバから返す値 */
61:     char far *id_ptr;             /* ボリュームIDへのポインタ */
62: } MDACHK_PACKET;
63:
```

〔リストC-5〕ヘッダ・ファイル struc.h ②

```
64: /*************************************************************************
65: *
66: *     構造体：   _BLDBPB_PACKET
67: *     機  能：   BUILD BPB コマンド用コマンド・パケットの定義
68: *
69: *************************************************************************/
70: typedef struct _BLDBPB_PACKET {
71:     REQ_HEAD packet;              /* リクエスト・ヘッダ */
72:     unsigned char media_dsc;      /* メディア・ディスクリプタ */
73:              char far *trans;     /* 転送アドレス */
74:     BPB  far *bpb_adrs;           /* ＢＰＢへのポインタ */
75: } BLDBPB_PACKET;
76:
77: /*************************************************************************
78: *
79: *     構造体：   _RW_PACKET
80: *     機  能：   READ/WRITE コマンド用コマンド・パケットの定義
81: *
82: *************************************************************************/
83: typedef struct _RW_PACKET {
84:     REQ_HEAD packet;              /* リクエスト・ヘッダ */
85:     unsigned char media_dsc;      /* メディア・ディスクリプタ */
86:              char far *trans;     /* 転送アドレス */
87:     unsigned int  sct_count;      /* 転送セクタ数 */
88:     unsigned int  sct_begin;      /* 開始セクタ */
89:              char far *id_ptr;    /* ボリュームＩＤへのポインタ */
90: } RW_PACKET;
```

● ramc.c モジュール

リストC-6のモジュールは，RAMディスク・ドライバのコマンド・コードの解析と処理を行います．

このファイルでは，グローバル変数名の最初の1文字に大文字を用いて，関数内で変数をアクセスする際にローカル変数とグローバル変数の区別が明確にできるようにしています．

構造体変数Bpbは，BPBテーブルの確保と初期化を行います．

ここでは，セクタ・サイズを1024バイトに，1クラスタは1セクタで構成することにします．また，RAMディスクからシステムをブートすることはないので，システムの予約領域は0としています．

FATテーブルは，通常のディスク・ドライブと同様に2を指定しています．また，ルート・ディレクトリの数も196であり，1Mバイト・フォーマットのディスクと同じにしてあります．

ドライブ容量のフィールドには，デフォルト値の2048セクタ(2Mバイト)を設定しておき，もし，ドライブ登録時にサイズ・オプションが指定されていたら，ドライバのINITコマンド処理時に書き換えることにします．

次にメディア・ディスクリプタ・バイトには，1Mフォーマット・ディスクのFAT-IDの値(FEH)を用いることにします．そして，FATテーブルのサイズは4セクタにしておきます．

このファイルでは，グローバル変数名の最初の1文字に大文字を用いて，関数内で変数をアクセスする際にローカル変数とグローバル変数の区別が明確にできるようにしています．

関数startは，割り込みエントリ・ルーチンから渡されたコマンド・パケットへのポインタを用いて，各デバイス・コマンドの処理を行います．

この関数では，switch文を用いてデバイス・コマンド・コードを調べ，各コマンド・コードに対応した関数を呼び出してデバイス・コマンドの処理を行っています．各関数からは，戻り値としてステータス・コードが返されるので，その値をコマンド パケットのステータス・フィールドに設定して返します．

もし，コマンド・コードが定義されているコード以外の場合は，ステータス・フィールドに8003H(無効なコマンド・コード)を返しています．

関数initはデバイス・コマンドのINITコマンドを処理します．この関数では，EMMに対してRAMディスク・ドライバで使用する拡張メモリを要求し，その拡張メモリのディレクトリ領域やFAT領域を初期化しておきます．これによって，RAMディスク用の特殊なフォーマット・コマンドのたぐいは必要なくなります．

まず，最初に関数func_01を用いてEMMから返されたステータスを調べ，EMMがメモリに常駐していることを確認しています．

次に，このドライバはEMM ver.4.0で拡張されたEMMファンクションを使用しているため，関数func_07を用いてEMMのバージョン番号を調べます．ここでは，EMMドライバの確認を簡単な手続きで済ま

せていますが，前出のget interrupt vector法を用いて正式に調べるべきでしょう．

次に，関数sdspを用いてRAMディスク・ドライバのタイトルを表示します．タイトルの表示につづいてコマンド・パケットに渡されたデバイス番号を用いてドライブ名の表示を行います．

そして，関数func_03を用いて拡張メモリのサイズを読み出し，その表示を行います．

さらに，関数read_capを用いてデバイス登録時のサイズ・オプションの読み出しを行います．得られたディスク容量をもとに関数func_04によって拡張メモリの割り当てを要求します．そして，そのディスク容量やページ数の表示を行います．

これらの処理が終わったら，コマンド・パケット内のドライブ数フィールドの設定を行います．

そして次に，BPBテーブルのディスク容量フィールドの設定を行い，コマンド・パケットのBPB配列へのポインタのフィールドやブレーク・アドレスのフィールドを設定します．ここで，BPBテーブル配列へのポインタ・フィールドにBPBテーブルへの直接のポインタを設定してはいけません．

また，ブレーク・アドレスのフィールドには，セグメント_dend内に定義しておいたラベルd_endのアドレスを設定します．

次に，関数set_fatを用いてFATテーブルの初期化を行って，関数set_fatからの戻り値をそのまま関数initの戻り値として返しています．

関数media_checkは，MEDIA CHECKコマンドの処理を行います．RAMディスクでは，メディアの交換を行うケースがないため，ここでは常に"1"(交換していない)を返しています．

関数bld_bpbは，BUILD BPBコマンドの処理を行います．RAMディスクでは，メディアの交換をすることがないためBPBテーブルも選択する余地はありません．したがって，この関数では，コマンド・パケットのBPBテーブルへのポインタのフィールドに対して，常にグローバル変数Bpbへのポインタを設定して返しています．

ここで，コマンド・パケット内のBPBへのポインタ・フィールドは，関数initでのBPB配列へのポインタではないことに注意が必要です．

関数readは，READコマンドの処理を行います．前述したように，デバイス・ドライバが呼び出された時点において，アプリケーション・プログラムですでに拡張メモリを使用している可能性があります．このため，デバイス・ドライバでは現在の拡張メモリのマッピング状態を保存しなければなりません．

関数readでは，まず関数func_1500を用いてページ・マップの退避を行います．次に，コマンド・パケットで指定されたセクタ番号がディスク容量を越えていないかどうかを調べ，もし越えている場合には，戻り値として8008H(セクタが存在しない)を設定してリターンします．

そして，関数func_02を用いて物理ページのセグメント・アドレスを得ます．さらにdo〜whileループと関数func_05を用いて，指定されたセクタに対応した論理ページを物理ページにマッピングし，実際のデータ転送は関数read_subに任せています．これらの処理はセクタの数だけ繰り返して処理されます．

すべてのセクタのデータ転送が終了したら，関数func_1501を用いてマッピング状態の復元を行い，戻り値として0100H(エラーなし終了)を返します．

関数writeは，関数readとほとんど同じ処理を行います．ここで，関数readとの違いはデータの転送方向だけであり，関数readがデータ転送を関数read_subに任せるのに対し，この関数では関数write_subを用いて行います．

関数set_fatは，RAMディスクのFATテーブルの初期化を行います．この関数でも拡張メモリのマッピングを行うために，まず，関数func_1500によって現在のマッピング状態の保存を行います．次に関数func_05を用いてFATセクタに対応した拡張メモリの論理ページをマッピングします．

そして，関数func_02で得られた物理ページのセグメント・アドレスと関数fat_initを用いてFATセクタの初期化を行います．FATセクタの初期化が終了したら関数func_1501を用いてマッピング状態をもとの状態に復元しておきます．

関数read_capは，デバイス登録時のオプションで指定されたディスク容量の読み出しを行います．この関数では，最初のNULLキャラクタ(空白コードが置き換えられる)に出会うまでポインタを進め，それ以降の文字列(数字)をバッファに格納し，その数字列を関数atoiに渡して数値に変換します．そして，得られた数値を戻り値として返します．

関数emm_errは，EMMファンクションのアクセス時にエラーが発生した場合にそのエラー処理を行います．ドライバ内でEMMアクセス時にエラーが発生したまま終了すると，ほかのアプリケーション・プログラムで拡張メモリを使用できなくなるため，関数func_06を用いてドライバで使用している拡張メモリをEMMに返却し，戻り値として800CHを返しています．

関数ddspは，与えられた数値を10進文字列に変換してスクリーンに表示します．この関数は，ディスク容量や拡張メモリのページ数を表示する際にコールさ

れます．まず，関数 itoa を用いて与えられた数値をローカル・バッファに数字列として変換します．次に，変換された数字列を関数 sdsp によってスクリーンに表示します．

関数 atoi は，ポインタで渡された文字列を数値に変換します．変換された数値(ワード値)は戻り値として返されます．

関数 itoa は，与えられた int 変数を文字列に変換します．この関数では，基数を引数として与えることができます．変換された文字列は指定されたバッファに格納されます．

関数 strlen は，引数で指定された文字列の長さ(バイト)を調べて，その値を戻り値として返します．

関数 strrev は，関数 itoa からコールされて文字列を逆に並べます．関数 itoa では，数値を数字に変換する際に下位の桁から変換していくために，上位と下位の数字列が逆順に並べられます．このため，関数 strrev を用いて文字列を並べ替えています．

関数 isdigit は，渡された文字コードが 10 進数字の文字コードであるかどうかを調べます．もし，10 進数字の場合は真(ゼロ以外：TRUE)を返し，それ以外の文字コードの場合は偽(ゼロ：FALSE)を返します．

関数 isspace は，渡された文字コードが空白の文字コードであるかどうかを調べます．もし，空白文字の場合は真(ゼロ以外：TRUE)を返し，それ以外の文字コードの場合は偽(ゼロ：FALSE)を返します．

〔リストC-6〕 プログラム ramc.c ①

```
 1: /*****************************************************************************
 2: *
 3: *    機    能：  拡張メモリ対応ＲＡＭディスク・ドライバ（コマンド部分）
 4: *    生    成：  cl -AS -c ramc.c
 5: *
 6: *****************************************************************************/
 7: #include      "struc.h"
 8: #define       DFLT_SIZE        2048
 9: #define       PAGE_SIZE        16
10: #define       SCT_SIZE         1024
11: #define       STR_LEN          10
12: #define       BUFF_LEN         256
13: #define       TRUE             1
14: #define       FALSE            0
15:
16: /*****************************************************************************
17: *
18: *    構造体：  Bpb
19: *    機  能：  BPB テーブルの確保と初期化
20: *
21: *****************************************************************************/
22: BPB Bpb = {
23:     SCT_SIZE, 1, 0, 2, 196, DFLT_SIZE, 0xFE, 4
24: };
25:
26: /*****************************************************************************
27: *
28: *    機    能：   グローバル変数の宣言
29: *
30: *****************************************************************************/
31: unsigned int Reg_ax;
32: unsigned int Reg_bx;
33: unsigned int Reg_dx;
34: BPB far *Bpb_ptr;
35: int Emm_ptr, Cap;
36:
37: /*****************************************************************************
38: *
39: *    機    能：   ＡＮＳＩプロトタイプ関数宣言
40: *
41: *****************************************************************************/
42: void start (REQ_HEAD far *);
43: int  init (INIT_PACKET far *);
44: int  media_check (MDACHK_PACKET far *);
45: int  bld_bpb (BLDBPB_PACKET far *);
46: int  read (RW_PACKET far *);
47: int  write (RW_PACKET far *);
48: int  set_fat (int);
49: int  read_cap (char far *);
50: int  emm_err (void);
```

〔リストC-6〕プログラム ramc.c ②

```
 51: void ddsp (int);
 52: char *strrev (char *);
 53: int  atoi (char *);
 54: char *itoa (int, char*, int);
 55: int  strlen (char *);
 56: int  isdigit (int);
 57: int  isspace (int);
 58:
 59: int  func_01 (void);
 60: int  func_02 (void);
 61: int  func_03 (void);
 62: int  func_04 (int);
 63: int  func_05 (int, int, int);
 64: int  func_06 (int);
 65: int  func_07 (void);
 66: int  func_1500 (char *);
 67: int  func_1501 (char *);
 68: char far * read_sub (int, int, int, char far *);
 69: char far * write_sub (int, int, int, char far *);
 70: void fat_init (int, int, int);
 71: void sdsp (char *);
 72:
 73: /*****************************************************************
 74: *
 75: *   関数名 :  start ()
 76: *   機  能 :  コマンド・コードの解析
 77: *   入  力 :  ptr     コマンド・パケットへのポインタ
 78: *   出  力 :  なし
 79: *
 80: *****************************************************************/
 81: void start (ptr)
 82: REQ_HEAD far *ptr;
 83: {
 84:     int ret_code;
 85:
 86:     switch (ptr -> cmd_code) {
 87:     case 0:
 88:         ret_code = init ((INIT_PACKET far *)ptr);
 89:         break;
 90:     case 1:
 91:         ret_code = media_check ((MDACHK_PACKET far *)ptr);
 92:         break;
 93:     case 2:
 94:         ret_code = bld_bpb ((BLDBPB_PACKET far *)ptr);
 95:         break;
 96:     case 4:
 97:         ret_code = read ((RW_PACKET far *)ptr);
 98:         break;
 99:     case 8:
100:     case 9:
101:         ret_code = write ((RW_PACKET far *)ptr);
102:         break;
103:     default:
104:         ptr -> status = 0x8003;
105:         return;
106:     }
107:     ptr -> status = ret_code;
108: }
109:
110: /*****************************************************************
111: *
112: *   関数名 :  init (ptr)
113: *   機  能 :  INIT コマンド
114: *   入  力 :  ptr   コマンド・パケットへのポインタ
115: *   出  力 :  ステータス・コード
116: *
117: *****************************************************************/
118: int init (ptr)
119: INIT_PACKET far *ptr;
120: {
121:     char buff[STR_LEN];
122:     extern int d_end;
123:
124:     if (func_01 ()) {                           /* ステータスの取得  */
125:         sdsp ("拡張メモリ・ドライバをインストールして下さい.￥n");
126:         return (0x800c);
```

〔リストC-6〕プログラム ramc.c ③

```
127:    }
128:    func_07 ();                                     /* バージョン番号 */
129:    if (Reg_ax != 0x0040) {
130:        sdsp ("拡張メモリ・ドライバのバージョンが違います.¥n");
131:        return (0x800c);
132:    }
133:    sdsp ("¥n拡張メモリ対応RAMディスク・ドライバ Ver.1.1");
134:    sdsp ("  programed by H.Abe¥n");
135:    sdsp ("              RAMディスクのドライブ名  ");
136:    buff [0] = ptr -> dev_num + 'A'; buff [1] = '¥0';
137:    sdsp (buff); sdsp (" ドライブ¥n");
138:    func_03 ();                                     /* 未アロケート・ページ数の取得 */
139:    sdsp ("              拡張メモリの総ページ数  : ");
140:    ddsp (Reg_dx); sdsp (" ページ¥n");
141:    Cap = read_cap ((char far *)ptr -> bpb);        /* ディスク容量 */
142:    if (!Cap) {
143:        Cap = DFLT_SIZE;
144:    }
145:    if (Reg_bx * PAGE_SIZE < Cap) {
146:        Cap = Reg_bx * PAGE_SIZE;                   /* 割り当てられたキロ・バイト数 */
147:    }
148:    if (func_04 (Cap / PAGE_SIZE)) {                /* ページの割り当て */
149:        return (emm_err ());
150:    }
151:    Emm_ptr = Reg_dx;
152:    sdsp ("              RAMディスクのページ数  : ");
153:    ddsp (Cap / PAGE_SIZE); sdsp (" ページ(16Kバイト単位)¥n");
154:    sdsp ("              RAMディスクの容量      : ");
155:    ddsp (Cap); sdsp (" Kバイト¥n");               /* 容量表示 */
156:    ptr -> unit = 1;                                /* ユニット数 */
157:    Bpb.cap = Cap;
158:    Bpb_ptr = (BPB far *)&Bpb;                      /* BPBへのポインタ */
159:    ptr -> bpb = (BPB far *) &Bpb_ptr;              /* BPB配列へのポインタ */
160:    ptr -> end = (char far *)&d_end;                /* 最終ドレス */
161:    return (set_fat (Cap));
162: }
163:
164: /****************************************************************
165:  *
166:  *   関数名 :  media_check (ptr)
167:  *   機  能 :  MEDIA CHECK コマンド
168:  *   入  力 :  ptr ‥‥ コマンド・パケットへのポインタ
169:  *   出  力 :  ステータス・コード
170:  *
171:  ****************************************************************/
172: int media_check (ptr)
173: MDACHK_PACKET far *ptr;
174: {
175:    ptr -> ret_code = 1;                   /* 交換なし */
176:    return (0x100);
177: }
178:
179: /****************************************************************
180:  *
181:  *   関数名 :  bld_bpb (ptr)
182:  *   機  能 :  BUILD BPB コマンド
183:  *   入  力 :  ptr ‥‥ コマンド・パケットへのポインタ
184:  *   出  力 :  ステータス・コード
185:  *
186:  ****************************************************************/
187: int bld_bpb (ptr)
188: BLDBPB_PACKET far *ptr;
189: {
190:    ptr -> bpb_adrs = (BPB far *) &Bpb;    /* BPBへのポインタ */
191:    return (0x100);
192: }
193:
194: /****************************************************************
195:  *
196:  *   関数名 :  read (ptr)
197:  *   機  能 :  READ コマンド
198:  *   入  力 :  ptr ‥‥ コマンド・パケットへのポインタ
199:  *   出  力 :  ステータス・コード
200:  *
201:  ****************************************************************/
202: int read (ptr)
```

〔リストC-6〕プログラム ramc.c ④

```
203: RW_PACKET far *ptr;
204: {
205:     char buff [BUFF_LEN];
206:     unsigned int sct_n, sct_bgn;
207:     unsigned int page, off, seg;
208:
209:     sct_n = ptr -> sct_count;
210:     sct_bgn = ptr -> sct_begin;
211:     if (func_1500 (buff)) {                    /* ページ・マップの退避 */
212:         return (emm_err ());
213:     }
214:     if (sct_bgn + sct_n > Cap) {
215:         return (0x8008);                       /* セクタが存在しない */
216:     }
217:     func_02 (); seg = Reg_bx;                  /* ページ・フレームのセグメント */
218:     do {
219:         page = sct_bgn / PAGE_SIZE;                    /* ページ番号 */
220:         off = (sct_bgn % PAGE_SIZE) * SCT_SIZE;        /* ページ内のオフセット */
221:         if (func_05 (0, page, Emm_ptr)) {              /* ページのマップ */
222:             return (emm_err ());
223:         }
224:         ptr -> trans = read_sub (SCT_SIZE, seg, off, ptr -> trans);
225:         sct_bgn ++;
226:     } while (--sct_n);                         /* セクタ数だけ繰り返し */
227:     if (func_1501 (buff)) {                    /* ページ・マップの復帰 */
228:         return (emm_err ());
229:     }
230:     return (0x100);
231: }
232:
233: /**************************************************************
234: *
235: *    関数名：  write (ptr)
236: *    機　能：  READ コマンド
237: *    入　力：  ptr ……… コマンド・パケットへのポインタ
238: *    出　力：  ステータス・コード
239: *
240: ***************************************************************/
241: int write (ptr)
242: RW_PACKET far *ptr;
243: {
244:     char buff [BUFF_LEN];
245:     unsigned int sct_n, sct_bgn;
246:     unsigned int page, off, seg;
247:
248:     sct_n = ptr -> sct_count;
249:     sct_bgn = ptr -> sct_begin;
250:     if (func_1500 (buff)) {                    /* ページ・マップの退避 */
251:         return (emm_err ());
252:     }
253:     if (sct_bgn + sct_n > Cap) {
254:         return (0x8008);                       /* セクタが存在しない */
255:     }
256:     func_02 (); seg = Reg_bx;                  /* ページ・フレームのセグメント */
257:     do {
258:         page = sct_bgn / PAGE_SIZE;                    /* ページ番号 */
259:         off = (sct_bgn % PAGE_SIZE) * SCT_SIZE;        /* ページ内のオフセット */
260:         if (func_05 (0, page, Emm_ptr)) {              /* ページのマップ */
261:             return (emm_err ());
262:         }
263:         ptr -> trans = write_sub (SCT_SIZE, seg, off, ptr -> trans);
264:         sct_bgn ++;
265:     } while (--sct_n);                         /* セクタ数だけ繰り返し */
266:     if (func_1501 (buff)) {                    /* ページ・マップの復帰 */
267:         return (emm_err ());
268:     }
269:     return (0x100);
270: }
271:
272: /**************************************************************
273: *
274: *    関数名：  set_fat (n)
275: *    機　能：  FAT の初期化
276: *    入　力：  n          セクタ数
277: *    出　力：  ステータス・コード
```

〔リストC-6〕プログラム ramc.c ⑤

```
278: *
279: ***************************************************************/
280: int set_fat (n)
281: int n;
282: {
283:     unsigned int seg;
284:     char buff [BUFF_LEN];
285:     int ret_code;
286:
287:     if (func_1500 (buff))           /* ページ・マップの退避 */
288:         return (emm_err ());
289:     }
290:     if (func_05 (0, 0, Emm_ptr)) {  /* ハンドル・ページのマップ */
291:         return (emm_err ());
292:     }
293:     func_02 ();                     /* フレーム・アドレスの取得 */
294:     seg = Reg_bx;                   /* フレーム・セグメント */
295:     fat_init (SCT_SIZE * PAGE_SIZE, Bpb.dev_disc, seg);
296:     if (ret_code = func_1501 (buff)) {      /* ページ・マップの復帰 */
297:         return (emm_err ());
298:     }
299:     return (0x0100);
300: }
301:
302: /**************************************************************
303: *
304: *    関数名 :  read_cap (ptr)
305: *    機  能 :  オプション(ディスク容量)の読み出し
306: *    入  力    ptr ・・ 文字列へのポインタ
307: *    出  力    容量
308: *
309: ***************************************************************/
310: int read_cap (ptr)
311: char far *ptr;
312: {
313:     char buff [BUFF_LEN];
314:     char *str = buff;
315:
316:     if (*ptr == 0x0D) {
317:         return (0);
318:     }
319:     while (*ptr != '¥0') {
320:         ptr++;
321:     }
322:     while (*++ptr != 0x0D) {
323:         *str++ = *ptr;
324:     }
325:     *str = '¥0';
326:     return (atoi (buff));
327: }
328:
329: /**************************************************************
330: *
331: *    関数名 :  emm_err ()
332: *    機  能 :  エラー処理
333: *    入  力 :  なし
334: *    出  力 :  ステータス・コード(エラー・コード)
335: *
336: ***************************************************************/
337: int emm_err ()
338: {
339:     sdsp ("¥n拡張メモリのアクセス中にエラーが発生しました.¥n");
340:     func_06 (Emm_ptr);
341:     return (0x800C);
342: }
343:
344: /**************************************************************
345: *
346: *    関数名 :  ddsp (n)
347: *    機  能 :  10進表示
348: *    入  力 :  n ・・・ 16進整数
349: *    出  力 :  なし
350: *
351: ***************************************************************/
352: void ddsp (n)
```

〔リストC-6〕プログラムramc.c ⑥

```
353: int n;
354: {
355:     char buff[STR_LEN];
356: 
357:     itoa (n, buff, 10);
358:     sdsp (buff);
359: }
360: 
361: /**************************************************************
362: *
363: *   関数名 :  atoi (s)
364: *   機  能 :  文字列 → int への変換
365: *   入  力 :  s ・・・ 文字列へのポインタ
366: *   出  力 :  ワード整数
367: *
368: **************************************************************/
369: int atoi (s)
370: char *s;
371: {
372:     int sign;
373:     int i;
374: 
375:     while (isspace (*s)) {
376:         s++;
377:     }
378:     sign = 1;
379:     switch (*s) {
380:     case '-':
381:         sign = -1;
382:     case '+':
383:         s++;
384:         break;
385:     }
386:     for (i = 0; isdigit (*s); ++s) {
387:         i = 10 * i + (*s - '0');
388:     }
389:     return (sign * i);
390: }
391: 
392: /**************************************************************
393: *
394: *   関数名 :  itoa (val, s, radix)
395: *   機  能 :  int → 文字列への変換
396: *   入  力 :  val    ・・・・・ int変数
397: *             s      ・・・・・ バッファへのポインタ
398: *             radix  ・・・・・ 基数
399: *   出  力 :  文字列へのポインタ
400: *
401: **************************************************************/
402: char *itoa (val, s, radix)
403: int val;
404: char *s;
405: int radix;
406: {
407:     char *s1;
408:     int sign;
409: 
410:     s1 = s;
411:     if ((radix == 10) && (val < 0)) {
412:         sign = 1;
413:         val = -val;
414:     } else {
415:         sign = 0;
416:     }
417:     do {
418:         *s = val % radix;
419:         *s += (*s <= 9 ? '0': ('A' - 10));
420:         ++s;
421:     } while (val /= radix);
422:     if (sign) {
423:         *s++ = '-';
424:     }
425:     *s = '¥0';
426:     strrev (s1);
427:     return (s1);
428: }
429: 
```

〔リストC-6〕プログラム ramc.c ⑦

```
430: /*****************************************************************
431: *
432: *   関数名 :  strlen (str)
433: *   機  能 :  文字列の長さを調べる
434: *   入  力 :  str ・・・・ 文字列へのポインタ
435: *   出  力 :  文字列の長さ
436: *
437: *****************************************************************/
438: int strlen (str)
439: char *str;
440: {
441:     int i;
442:
443:     for (i = 0; *str; str++) {
444:         ++i;
445:     }
446:     return (i);
447: }
448:
449: /*****************************************************************
450: *
451: *   関数名 :  strrev (str)
452: *   機  能 :  文字列を逆に並べる
453: *   入  力 :  str ・・・・ 文字列へのポインタ
454: *   出  力 :  文字列へのポインタ
455: *
456: *****************************************************************/
457: char *strrev (str)
458: char *str;
459: {
460:     char *s, *s1, c;
461:     int n;
462:
463:     s1 = str;
464:     n = strlen (str);
465:     s = str + n - 1;
466:     n /= 2;
467:     while (n--) {
468:         c = *str;
469:         *str++ = *s;
470:         *s-- = c;
471:     }
472:     return (s1);
473: }
474:
475: /*****************************************************************
476: *
477: *   関数名 :  isdigit (c)
478: *   機  能 :  10進数字のチェック
479: *   入  力 :  s ・・・・ 文字コード
480: *   出  力 :  10進数字 : TRUE      それ以外 : FALSE
481: *
482: *****************************************************************/
483: int isdigit (c)
484: int c;
485: {
486:     if (c >= '0' && c <= '9') {
487:         return (TRUE);
488:     }
489:     return (FALSE);
490: }
491:
492: /*****************************************************************
493: *
494: *   関数名 :  isspace (c)
495: *   機  能 :  空白コードのチェック
496: *   入  力 :  s ・・・・ 文字コード
497: *   出  力 :  空白文字 : TRUE      それ以外 : FALSE
498: *
499: *****************************************************************/
500: int isspace (c)
501: int c;
502: {
503:     if (c == ' ') {
504:         return (TRUE);
505:     }
506:     return (FALSE);
507: }
```

● ramsub.asm モジュール

リストC-7 のモジュールは，RAM ディスク・ドライバのアセンブリ言語サブルーチンで，主として CPU レジスタの設定や読み出し，および EMM ファンクションの呼び出しを行っています．

CPU レジスタの操作や EMM ファンクションの呼び出しは，C 言語のみによっても記述可能です．しかし，レジスタ操作に関しては，アセンブリ言語のほうがレジスタとパラメータの対応が取りやすいため，あえてアセンブリ言語で記述してあります．

同リストにおいて，ディレクティブ EXTRN で宣言されているワード値のラベルは，**リストC-6** の ramc.c モジュール内で確保しているグローバル変数であり，EMM ファンクションやその他のシステム・コールの結果返されたレジスタ値を格納するのに使用されます．

サブルーチン(関数)func_01 は，EMM ファンクション 01H を用いて EMM ステータスの取得を行います．得られた EMM ステータスは戻り値として AX レジスタに返しています．

サブルーチン(関数)func_02 は，EMM ファンクション 02H を用いて物理ページのフレーム・アドレスを取得します．EMM ファンクションから返されたステータス・コードは戻り値として AX レジスタに返しています．また，得られたフレーム・アドレスは，グローバル変数 Reg_bx に設定しています．

サブルーチン(関数)func_03 は，EMM ファンクション 03H を用いて拡張メモリの未アロケートのページ数を取得します．EMM ファンクションから返されたステータス・コードは戻り値として AX レジスタに返しています．

また，得られた未アロケートのページ数は，グローバル変数 Reg_bx に設定し，グローバル変数 Reg_dx には拡張メモリの総ページ数を設定しています．

サブルーチン(関数)func_04 は，EMM ファンクション 04H を用いて拡張メモリのページ割り当てを要求します．EMM ファンクションから返されたステータス・コードは戻り値として AX レジスタに返しています．また，グローバル変数 Reg_dx には，EMM から返された EMM ハンドルを設定して返します．

サブルーチン(関数)func_05 は，EMM ファンクション 05H を用いて拡張メモリのページをマッピングします．EMM ファンクションから返されたステータス・コードは戻り値として AX レジスタに返しています．

サブルーチン(関数)func_06 は，EMM ファンクション 06H を用いて拡張メモリからドライバに対して割り当てられているページの解放を行います．EMM ファンクションから返されたステータス・コードは戻り値として AX レジスタに返しています．

サブルーチン(関数)func_07 は，EMM ファンクション 07H を用いて EMM バージョン番号の取得を行います．EMM ファンクションから返されたステータス・コードは戻り値として AX レジスタに返しています．また，得られたバージョン番号は，グローバル変数 Reg_ax に設定して返します．

サブルーチン(関数)func_1500 は，EMM ファンクション 1500H を用いてページ・マップの退避を行います．EMM ファンクションから返されたステータス・コードは戻り値として AX レジスタに返しています．

サブルーチン(関数)func_1501 は，EMM ファンクション 1501H を用いて退避しておいたページ・マップの復帰を行います．EMM ファンクションから返されたステータス・コードは戻り値として AX レジスタに返しています．

サブルーチン(関数)read_sub は，RAM ディスクのセクタ番号に対応した物理ページから 1 セクタ分のデータを読み込み，指定されたアドレスに転送します．このサブルーチンでは，戻り値として DX：AX レジスタに次の転送アドレスを FAR ポインタとして返しています．

サブルーチン(関数)write_sub は，指定されたアドレスから RAM ディスクのセクタ番号に対応した物理ページに対して，1 セクタ分のデータを転送します．このサブルーチンでも，戻り値として DX：AX レジスタに次の転送アドレスを FAR ポインタとして返しています

サブルーチン(関数)fat_init は，FAT セクタの初期化を行います．ここでは，指定されたバイト数だけのメモリ・クリアを行い，指定されたデバイス・ディスクリプタを FAT-ID フィールドに設定しています．

サブルーチン(関数)sdsp は，引数で渡されたポインタの指す文字列をスクリーンに表示します．ここで扱う文字列は ASCIZ 文字列である必要があります．

● st.asm モジュール

リストC-8 は，RAM ディスク・ドライバをデバッグするためのメイン・モジュールの記述例を示しています．このモジュールからは，**リストC-1**(ram.mak)の MAKE ファイルによって自動的にアセンブル/コンパイルおよびリンク処理が行われ，結果として stram.exe が実行モジュールとして生成されます．これによって，デバイス・ドライバの処理部分を CodeView を用いてデバッグすることが可能となります．

このモジュールでは，MS-DOS になったつもりでデバイス・ドライバを呼び出していかなければなりません．したがって，リクエスト・ヘッダを含むコマンド・

〔リストC-7〕プログラム ramsub.asm ①

```
 1: ;**********************************************************************
 2: ;
 3: ;    機    能:  拡張メモリのアクセスとデータ転送
 4: ;    生    成:  masm /ML ramsub;
 5: ;
 6: ;**********************************************************************
 7:             MODEL  SMALL, C
 8: ;**********************************************************************
 9: ;
10: ;    機    能:   外部参照
11: ;
12: ;**********************************************************************/
13:             EXTRN    Reg_ax:WORD, Reg_bx:WORD, Reg_dx:WORD
14:             CODE
15: ;**********************************************************************
16: ;
17: ;    ルーチン名 :    func_01
18: ;    機   能 : ステータスの取得
19: ;    EMM     :    01
20: ;    入   力 : なし
21: ;    出   力 : AX        ・・・ ステータス・コード
22: ;
23: ;**********************************************************************
24: func_01     PROC
25:             mov      ah, 40h
26:             int      67h                 ;ステータスの取得
27:             mov      al, ah              ;ステータス・コード
28:             xor      ah, ah
29:             ret
30: func_01     ENDP
31:
32: ;**********************************************************************
33: ;
34: ;     ルーチン名 :    func_02
35: ;     機   能 :    ページ・フレームのアドレス取得
36: ;     EMM     :    02
37: ;     入   力 :    なし
38: ;     出   力 :    AX     ・・・ ステータス・コード
39: ;                  Reg_bx ・・・ ページ・フレームのセグメント・アドレス
40: ;
41: ;**********************************************************************
42: func_02     PROC
43:             mov      ah, 41h
44:             int      67h                 ;ページ・フレームのアドレス取得
45:             mov      al, ah              ;ステータス・コード
46:             xor      ah, ah
47:             mov      Reg_bx, bx          ;セグメント・アドレス
48:             ret
49: func_02     ENDP
50:
51: ;**********************************************************************
52: ;
53: ;     ルーチン名 :    func_03
54: ;     機   能 :    未アロケート・ページ数の取得
55: ;     EMM     :    03
56: ;     入   力 :    なし
57: ;     出   力 :    AX     ・・・ ステータス・コード
58: ;                  Reg_bx ・・・ 未アロケートのページ数
59: ;                  Reg_dx ・・・ 総ページ数
60: ;
61: ;**********************************************************************
62: func_03     PROC
63:             mov      ah, 42h
64:             int      67h                 ;未アロケート・ページ数の取得
65:             mov      al, ah              ;ステータス・コード
66:             xor      ah, ah
67:             mov      Reg_bx, bx          ;未アロケートのページ数
68:             mov      Reg_dx, dx          ;総ページ数
69:             ret
70: func_03     ENDP
71:
72: ;**********************************************************************
73: ;
74: ;     ルーチン名 :    func_04
75: ;     機   能 :    ページのアロケート
76: ;     EMM     :    04
77: ;     入   力 :    arg1   ・・・ 要求するページ数
78: ;     出   力 :    AX     ・・・ ステータス・コード
```

〔リストC-7〕プログラム ramsub.asm ②

```
 79: ;           Reg_dx  ･･･ EMM ハンドル
 80: ;
 81: ;*******************************************************************
 82: func_04     PROC     arg1:WORD
 83:             mov      ah, 43h
 84:             mov      bx, arg1              ;割り当てるページ数
 85:             int      67h                   ;ページの割り当て
 86:             mov      al, ah                ;ステータス・コード
 87:             xor      ah, ah
 88:             mov      Reg_dx, dx            ;EMM ハンドル
 89:             ret
 90: func_04     ENDP
 91:
 92: ;*******************************************************************
 93: ;
 94: ;    ﾙｰﾁﾝ名 :  func_05
 95: ;    機  能 :  ハンドル・ページのマップ
 96: ;    EMM    :  05
 97: ;    入  力 :  arg1    ･･･ 物理ページ番号
 98: ;              arg2    ･･･ 論理ページ番号
 99: ;              arg3    ･･･ EMM ハンドル
100: ;    出  力 :  AX      ･･･ ステータス・コード
101: ;
102: ;*******************************************************************
103: func_05     PROC     arg1:WORD, arg2:WORD, arg3:WORD
104:             mov      ax, arg1              ;物理ページ番号
105:             mov      bx, arg2              ;論理ページ番号
106:             mov      dx, arg3              ;EMM ハンドル
107:             mov      ah, 44h
108:             int      67h                   ;ページのマップ
109:             mov      al, ah                ;ステータス・コード
110:             xor      ah, ah
111:             ret
112: func_05     ENDP
113:
114: ;*******************************************************************
115: ;
116: ;    ﾙｰﾁﾝ名 :  func_06
117: ;    機  能 :  割り当てられたページの解放
118: ;    EMM    :  06
119: ;    入  力 :  arg1    ･･･ EMM ハンドル
120: ;    出  力 :  AX      ･･･ ステータス・コード
121: ;
122: ;*******************************************************************
123: func_06     PROC     arg1:WORD
124:             mov      dx, arg1              ;物理ページ番号
125:             mov      ah, 45h
126:             int      67h                   ;ページの解放
127:             mov      al, ah                ;ステータス・コード
128:             xor      ah, ah
129:             ret
130: func_06     ENDP
131:
132: ;*******************************************************************
133: ;
134: ;    ﾙｰﾁﾝ名 :  func_07
135: ;    機  能 :  バージョン番号の取得
136: ;    EMM    :  07
137: ;    入  力 :  なし
138: ;    出  力 :  AX      ･･･ ステータス・コード
139: ;              Reg_ax  ･･･ バージョン番号
140: ;
141: ;*******************************************************************
142: func_07     PROC
143:             mov      ah, 46h
144:             int      67h                   ;バージョン番号の取得
145:             push     ax
146:             xor      ah, ah
147:             mov      Reg_ax, ax            ;バージョン番号
148:             pop      ax
149:             mov      al, ah                ;ステータス・コード
150:             xor      ah, ah
151:             ret
152: func_07     ENDP
153:
154: ;*******************************************************************
155: ;
156: ;    ﾙｰﾁﾝ名 :  func_1500
```

〔リストC-7〕プログラム ramsub.asm ③

```
157: ;    機  能 :    ページ・マップの取得(退避)
158: ;    EMM    :    15  コード00H
159: ;    入  力 :    arg1    ・・・バッファへのポインタ
160: ;    出  力 :    AX      ・・・ステータス・コード
161: ;
162: ;*****************************************************************
163: func_1500     PROC     USES ES DI, arg1:PTR
164:               mov      di, arg1
165:               push     ss
166:               pop      es
167:               mov      ax, 4E00h
168:               int      67h                  ;ページ・マップの取得
169:               mov      al, ah               ;ステータス・コード
170:               xor      ah, ah
171:               ret
172: func_1500     ENDP
173:
174: ;*****************************************************************
175: ;
176: ;    ルーチン名 :  func_1501
177: ;    機  能 :    ページ・マップの設定(復帰)
178: ;    EMM    :    1501
179: ;    入  力 :    arg1    ・・・バッファへのポインタ
180: ;    出  力 :    AX      ・・・ステータス・コード
181: ;
182: ;*****************************************************************
183: func_1501     PROC     USES SI, arg1:PTR
184:               mov      si, arg1             ;バッファへのポインタ
185:               mov      ax, 4E01h
186:               int      67h                  ;ページ・マップの設定
187:               mov      al, ah               ;ステータス・コード
188:               xor      ah, ah
189:               ret
190: func_1501     ENDP
191:
192: ;*****************************************************************
193: ;
194: ;    ルーチン名 :  read_sub
195: ;    機  能 :    ディスクの読み出し
196: ;    入  力 :    arg1    ・・・バイト数
197: ;                arg2    ・・・ページ・フレームのセグメント
198: ;                arg3    ・・・物理ページのオフセット
199: ;                arg4    ・・・転送アドレス
200: ;    出  力 :    DX:AX   ・・・転送バッファへのポインタ
201: ;
202: ;*****************************************************************
203: read_sub      PROC     USES DS ES DI SI,arg1:WORD,arg2:WORD,arg3:WORD,arg4:FAR PTR
204:               les      di, arg4             ;転送アドレス
205:               mov      si, arg3             ;物理ページ・オフセット
206:               mov      ds, arg2             ;物理ページ・セグメント
207:               mov      cx, arg1             ;転送バイト数
208:               shr      cx, 1                ;転送バイト数/2
209: read_loop:
210:               movsw                         ;ワード転送
211:               loop     read_loop
212:               mov      dx, es
213:               mov      ax, di
214:               ret
215: read_sub      ENDP
216:
217: ;*****************************************************************
218: ;
219: ;    ルーチン名 :  write_sub
220: ;    機  能 :    ディスクへの書き込み
221: ;    入  力 :    arg1    ・・・バイト数
222: ;                arg2    ・・・ページ・フレームのセグメント
223: ;                arg3    ・・・物理ページのオフセット
224: ;                arg4    ・・・転送アドレス
225: ;    出  力 :    DX:AX   ・・・転送バッファへのポインタ
226: ;
227: ;*****************************************************************
228: write_sub     PROC     USES DS ES DI SI,arg1:WORD,arg2:WORD,arg3:WORD,arg4:FAR PTR
229:               lds      si, arg4             ;転送アドレス
230:               mov      di, arg3             ;物理ページ・オフセット
231:               mov      es, arg2             ;物理ページ・セグメント
232:               mov      cx, arg1             ;転送バイト数
233:               shr      cx, 1                ;転送バイト/2
234: write_loop:
```

〔リストC-7〕プログラム ramsub.asm ④

```
235:            movsw                          ;ワード転送
236:            loop    write_loop
237:            mov     dx, ds
238:            mov     ax, si
239:            ret
240: write_sub  ENDP
241:
242: ;*************************************************************************
243: ;
244: ;     ルーチン名:  fat_init
245: ;     機　能:  FATの初期化
246: ;     入　力:  arg1     バイト数
247: ;              arg2     デバイス・ディスクリプタ
248: ;              arg3     セグメント・アドレス
249: ;     出　力:  なし
250: ;
251: ;*************************************************************************
252: fat_init       PROC    USES ES DI, arg1:WORD, arg2:WORD, arg3:WORD
253:               mov     es, arg3              ;ページ・フレーム・アドレス
254:               xor     di, di
255:               mov     cx, arg1              ;バイト数
256: clr_loop:
257:               mov     BYTE PTR es:[di], 0   ;FATクリア
258:               inc     di
259:               loop    clr_loop
260:               mov     ax, arg2
261:               xor     di, di
262:               mov     es:[di], al           ;FAT ID セット
263:               ret
264: fat_init       ENDP
265:
266: ;*************************************************************************
267: ;
268: ;     ルーチン名:  sdsp
269: ;     機　能:  文字列の表示
270: ;     入　力:  arg1  文字列へのポインタ
271: ;     出　力:  なし
272: ;
273: ;*************************************************************************
274: sdsp       PROC    arg1:PTR
275:            mov     bx, arg1                ;ポインタ
276:            mov     ah, 02h
277: sdsp_loop:
278:            mov     dl, [bx]
279:            or      dl, dl                  ;'¥0'(文字列の終端)  ?
280:            je      sdsp_exit
281:            cmp     dl, 0Ah                 ;LFコード?
282:            jne     sdsp_next
283:            mov     dl, 0Dh                 ;CRコード
284:            int     21h
285:            mov     dl, 0Ah
286: sdsp_next:
287:            int     21h
288:            inc     bx
289:            jmp     sdsp_loop
290: sdsp_exit:
291:            ret
292: sdsp       ENDP
293:            END
```

パケットを作成し，ストラテジ・ルーチンの呼び出し，割り込みエントリ・ルーチンの呼び出しを行います．

また，INIT コマンドの呼び出しでは，コマンド・パケットにオプション文字列へのポインタも正確に設定する必要があります．

このモジュールにおけるコード・セグメントは，リストC-2のセグメント名あるいはクラス名とは別名にしておく必要があります．もし，同名のセグメント名やクラス名を指定すると，ドライバ本体のコード・セグメントと一緒に扱われ，デバイス・ヘッダの配置が意図したとおりになりません．

〔リストC-8〕プログラム st.asm ①

```
 1: ;**************************************************************
 2: ;
 3: ;    機   能:  拡張メモリ対応ＲＡＭディスク・ドライバのデバッグ用
 4: ;    サ   ブ:  ram.asm asmsub.asm ramc.c
 5: ;    生   成:  make ramdbg.mak
 6: ;    使用方法:  cv ram
 7: ;
 8: ;**************************************************************/
 9:            PAGE    60, 130
10: SCT_SIZE   EQU     1024
11: SCT_BGN    EQU     120
12: SCT_N      EQU     63
13: DATA_SEG   EQU     9000h
14: DATA_OFF   EQU     0000h
15:
16: ;**************************************************************
17: ;
18: ;     構造体:  REQ_HEAD
19: ;     機  能:  リクエスト・ヘッダの定義
20: ;
21: ;**************************************************************/
22: REQ_HEAD     STRUC                          ;リクエスト・ヘッダ
23: pac_len      DB      ?                      ;パケット長
24: dev_code     DB      ?                      ;デバイス・コード
25: com_code     DB      ?                      ;コマンド・コード
26: status       DW      ?                      ;ステータス
27: reserve      DB      8 dup (?)              ;リザーブ
28: REQ_HEAD     ENDS
29:
30: ;**************************************************************
31: ;
32: ;     構造体:  INIT_PACKET
33: ;     機  能:  INIT コマンド用パケットの定義
34: ;
35: ;**************************************************************/
36: INIT_PACKET STRUC                           ;INIT コマンド用パケット
37: init_head    DB       TYPE REQ_HEAD DUP (?) ;リクエスト・ヘッダ
38: unit         DB       1                     ;ユニット数
39: dev_end      DD       ?                     ;エンド・アドレス
40: bpb          DD       ?                     ;ＢＰＢ配列へのポインタ
41: dev_num      DB       ?                     ;ドライブ番号
42: INIT_PACKET ENDS
43:
44: ;**************************************************************
45: ;
46: ;     構造体:  MDACHK_PACKET
47: ;     機  能:  MEDIA CHECK コマンド用パケットの定義
48: ;
49: ;**************************************************************/
50: MDACHK_PACKET    STRUC                      ;MEDIA CHECK 用パケット
51: mda_head         DB    TYPE REQ_HEAD DUP (?) ;リクエストヘッダ
52: mda_disc         DB    1                    ;メディア・デスクリプタ
53: ret_code         DB    ?                    ;ドライバから返す値
54: id_ptr           DD    ?                    ;ＩＤへのポインタ
55: MDACHK_PACKET    ENDS
56:
57: ;**************************************************************
58: ;
59: ;     構造体:  BLDBPB_PACKET
60: ;     機  能:  BULID BPB コマンド用パケットの定義
61: ;
62: ;**************************************************************/
63: BLDBPB_PACKET    STRUC                      ;BUILD BPB 用パケット
64: bpb_head         DB    TYPE REQ_HEAD DUP (?) ;リクエスト・ヘッダ
65: bpb_disc         DB    1                    ;メディア・デスクリプタ
66: bpb_trans        DD    ?                    ;転送アドレス
67: bpb_ptr          DD    ?                    ;ＩＤへのポインタ
68: BLDBPB_PACKET    ENDS
69:
70: ;**************************************************************
71: ;
72: ;     構造体:  RW_PACKET
73: ;     機  能:  READ / WRITE コマンド用パケットの定義
74: ;
75: ;**************************************************************/
76: RW_PACKET    STRUC                          ;BUILD BPB 用パケット
77: rw_head      DB       TYPE REQ_HEAD DUP (?) ;リクエスト・ヘッダ
78: rw_disc      DB       1                     ;メディア・デスクリプタ
```

〔リストC-8〕プログラム st.asm ②

```
 79: rw_trans      DD        ?                              ;転送アドレス
 80: sct_n         DW        ?                              ;セクタ・カウント
 81: sct_bgn       DW        ?                              ;開始セクタ
 82: rw_id         DD        ?                              ;IDへのポインタ
 83: RW_PACKET     ENDS
 84:
 85: ;**********************************************************************
 86: ;
 87: ;     SEGMENT:    データ・セグメント
 88: ;     機    能:   コマンド・パケットの確保
 89: ;
 90: ;**********************************************************************/
 91: _DATA1        SEGMENT WORD PUBLIC 'DATA1'
 92: config        DB        '¥wk1¥emm¥ram.sys', 0, '200', 0Dh, 0Ah, 00h
 93: init_com      INIT_PACKET        <>
 94: mdachk_com    MDACHK_PACKET      <>
 95: bldbpb_com    BLDBPB_PACKET      <>
 96: rd_com        RW_PACKET          <>
 97: wr_com        RW_PACKET          <>
 98: buff          DB        SCT_SIZE * SCT_N DUP (?)     ;セクタ・バッファ
 99: _DATA1        ENDS
100:
101: ;**********************************************************************
102: ;
103: ;     SEGMENT:    スタック・セグメント
104: ;     機    能:   スタックの確保
105: ;
106: ;**********************************************************************/
107: _STACK        SEGMENT WORD STACK 'STACK'
108: stack         DW        256 DUP (?)
109: _STACK        ENDS
110:
111: ;**********************************************************************
112: ;
113: ;     SEGMENT:    コード・セグメント
114: ;     機    能:   ドライバの呼び出し
115: ;
116: ;**********************************************************************/
117:               EXTRN     _strategy:FAR, _entry:FAR
118:               PUBLIC    begin
119:
120: _TEXT1        SEGMENT WORD PUBLIC 'TEXT1'
121:               ASSUME    CS:_TEXT1, DS:_DATA1, ES:_DATA1, SS:_STACK
122:
123: begin         PROC
124:               ;
125:               ; INITコマンドの呼び出し
126:               ;
127:               mov       ax, SEG init_com            ;コマンド・パケットのセグメント
128:               mov       es, ax
129:               lea       bx, init_com                ;コマンド・パケットのオフセット
130:               call      _strategy
131:               mov       es:[bx.pac_len], TYPE INIT_PACKET
132:               mov       es:[bx.com_code], 0
133:               lea       ax, config                  ;オプションのオフセット
134:               mov       WORD PTR es:[bx.bpb], ax
135:               mov       ax, SEG config              ;オプションのセグメント
136:               mov       WORD PTR es:[bx.bpb + 2], ax
137:               mov       es:[bx.dev_num], 'D' -'A'   ;ドライブ番号
138:               call      _entry                      ;INIT コマンド実行
139:               les       di, [bx.dev_end]            ;エンド・アドレス
140:               les       di, [bx.bpb]                ;BPB 配列へのポインタ
141:               les       di, [di]                    ;BPB へのポインタ
142:
143:               ;
144:               ; MEDIA CHECK コマンドの呼び出し
145:               ;
146:               mov       ax, SEG mdachk_com          ;コマンド・パケットのセグメント
147:               mov       es, ax
148:               lea       bx, mdachk_com              ;コマンド・パケットのオフセット
149:               call      _strategy
150:               mov       es:[bx.pac_len], TYPE MDACHK_PACKET
151:               mov       es:[bx.com_code], 1
152:               call      _entry                      ;MEDIA CHECK コマンド実行
153:               mov       al, [bx.ret_code]           ;返された値
154:
155:               ;
156:               ; BULD BPB コマンドの呼び出し
```

〔リストC-8〕プログラム st.asm ③

```
157:        ;
158:        mov     ax, SEG bldbpb_com          ;コマンド・パケットのセグメント
159:        mov     es, ax
160:        lea     bx, bldbpb_com              ;コマンド・パケットのオフセット
161:        call    _strategy
162:        mov     es:[bx.pac_len], TYPE BLDBPB_PACKET
163:        mov     es:[bx.com_code], 2
164:        call    _entry                      ;BUILD BPB コマンド実行
165:        les     di, [bx.bpb_ptr]            ;BPB へのポインタ
166:
167:        ;
168:        ; WRITE コマンドの呼び出し
169:        ;
170:        mov     ax, SEG wr_com              ;コマンド・パケットのセグメ
171:        mov     es, ax
172:        lea     bx, wr_com                  ;コマンド・パケットのオフセット
173:        call    _strategy
174:        mov     es:[bx.pac_len], TYPE RW_PACKET
175:        mov     es:[bx.com_code], 8
176:        mov     ax, DATA_OFF                ;転送アドレスのオフセット
177:        mov     WORD PTR es:[bx.rw_trans], ax
178:        mov     ax, DATA_SEG                ;転送アドレスのセグメント
179:        mov     WORD PTR es:[bx.rw_trans + 2], ax
180:        mov     es:[bx.sct_n], SCT_N        ;セクタ数
181:        mov     es:[bx.sct_bgn], SCT_BGN    ;開始セクタ
182:        call    _entry                      ;WRITE コマンド実行
183:        les     di, [bx.rw_trans]           ;転送アドレス
184:
185:        ;
186:        ; READ コマンドの呼び出し
187:        ;
188:        mov     ax, SEG rd_com              ;コマンド・パケットのセグメント
189:        mov     es, ax
190:        lea     bx, rd_com                  ;コマンド・パケットのオフセット
191:        call    _strategy
192:        mov     es:[bx.pac_len], TYPE RW_PACKET
193:        mov     es:[bx.com_code], 4
194:        lea     ax, buff                    ;転送アドレスのオフセット
195:        mov     WORD PTR es:[bx.rw_trans], ax
196:        mov     ax, SEG buff                ;転送アドレスのセグメント
197:        mov     WORD PTR es:[bx.rw_trans + 2], ax
198:        mov     es:[bx.sct_n], SCT_N / 2    ;セクタ数
199:        mov     es:[bx.sct_bgn], SCT_BGN    ;開始セクタ
200:        call    _entry                      ;WRITE コマンド実行
201:        les     di, [bx.rw_trans]           ;転送アドレス
202:
203:        mov     ax, 4C00h                   ;プログラム終了
204:        int     21h
205: begin  ENDP
206: _TEXT1 ENDS
207:        END     begin
```

ここでは，MS-DOSのキャラクタ・デバイスとブロック・デバイスについて，それぞれのドライバ実例を示しました．

グラフィック・コンソール・ドライバは，標準入力にグラフィック機能をもたせるための手法を示したもので，実用的にも価値の高いものです．このドライバをシステムに登録することによって，言語を問わずにグラフィックスを利用することができます．

このグラフィック・コンソール・ドライバは，ハード・コピー機能をもっていませんが，ハード・コピー機能をもたせるのはそれほど難しいことではないので，読者諸兄諸姉への宿題にしたいと思います．

拡張メモリを利用したRAMディスク・ドライバは，それほど珍しいものではありません．現に，あるメーカの拡張メモリ・ボードには，標準でRAMディスク・ドライバが添付されてきます．しかし，この程度のドライバなら，メーカの供給するドライバを使用するよりも，自分で開発したドライバを使用しているほうが気分的にも満足し，機能の追加や変更が自由に行えるので，ぜひ自分でプログラミングしていただきたいものです．

デバイス・ドライバは，一般のアプリケーションに比較して，約束ごとが多く構造も複雑なため，とっつきにくいものがあります．しかし，難しいが故に，プログラムが完成したときの感激にも大きなものがあります．

あとがき

これまで，MS-DOSのプログラム開発における各種ユーティリティの機能や，MS-DOS内部の構造まで，MS-DOSを活用してプログラム開発を行うための技術的な情報について解説してきました．

MS-DOSのように標準的で，よく普及しているOS上でプログラムを作成することは，プログラムの移植の際にも重要なポイントになります．

コンピュータ(ハードウェア)は，時代の流れとともにその機能が改善されて，新しいハードウェアに置き換えられていくことは確かです．しかし，一度開発されたユーザのソフトウェア(プログラム)は長持ちさせたいものです．

また，OSを活用することは，そのOSのコマンドや内部サブルーチンを利用することにより，プログラムの開発も効率よく行うことができます．同時に保守性もよくなります．

MS-DOSはUNIX流であるとはいえ，シングル・タスクであるが故に，その機能や使い勝手にはまだまだ低いものがありますが，パーソナル・ユースのプログラム開発環境としては十分に妥協のできる範囲にあるといえるでしょう．

マイクロコンピュータがこの世に登場した頃は，メモリ空間が狭かったのと，メモリ・コストも高かったために，プログラミングのウラ技的な手法が高い評価を受けていたものでした．小さいメモリで大きな仕事をさせるのが美徳とされていたのです．

しかし，今日のようにメモリ空間も広くなり，またメモリ・コストも下がってくると，今度は保守性のよい，汎用性のあるプログラムを作成することが重要な課題となってきました．

最近では，1本のプログラムを共同で開発することが多くなり，自分で記述したソース・リストの内容は，自分以外の人に対して文章のように適確に伝わらなければなりません．このため，ウラ技は陰を潜め，誰にでも読める/内容のわかるプログラミングが美徳になってきたのです．

そして，MS-DOS上で開発されたアプリケーションは，A社のパソコンでもB社のパソコンでも動かなければ意味がありません．処理速度も重要ですが，互換性/汎用性のほうがもっと重要だとは思いませんか．

本書の活用によって，1本でも多くのプログラムが開発され，それによって，1人でも多くのソフトウェア技術者が誕生することを念願するものです．そして，そのソフトウェア技術者たちが，互換性/汎用性を最重点にアプリケーションを開発してくれれば，我々ユーザとしては暮らしやすいパソコン社会になるのですが….

参考・引用*文献

(1) 田辺　正，8086マイクロコンピュータ，丸善．

(2) B.W.カーニハン他，プログラミング言語C，共立出版．

(3) *MS-DOS Ver.3.30ユーザーズ・マニュアル，日本電気．

(4) *マクロ・アセンブラ・パッケージVer.5.1マニュアルセット，日本電気．

(5) MS-C Ver.4.0マニュアルセット，マイクロソフト．

(6) PC9801と拡張インターフェース，トランジスタ技術SPECIAL No.3，CQ出版．

(7) 標準MS-DOSハンドブック，アスキー．

(8) MS-DOS 3.1ハンドブック，アスキー．

(9) MS-DOSプログラマーズ・ハンドブック，アスキー．

索引

〈ア〉
アセンブル……15
アドレス空間……30
アドレッシング・モード……52
〈イ〉
異常終了……95
イニシャル・プログラム・ローダ……83
〈ウ〉
ウインドウ……291
〈エ〉
エコー・バック……156
エディット……15
エラー・クラス……164
エラー・コード……155
エラー情報……131,284
演算子……76
〈オ〉
オーバライト……84
オーバレイ……160
オープン・ハンドル法……293
オフセット……30
オブジェクト・コード……35
オペランド……76
〈カ〉
開始セクタ番号……287
階層ディレクトリ……115,169
外部コマンド……88
外部参照……61
返す値……286
拡張エラー・コード……164
拡張メモリ・マネージャEMM……291
拡張FCB……123
仮想ドライブ……86
型演算子……78
カテゴリ・コード……288
カレント・ドライブ……122
カレント・ブロック……122,123
カレント・レコード……122,123
環境……88,97
環境変数……88,97
関係演算子……79
カントリ・コード……164
簡略化定義……55
〈キ〉
キーボードBIOS……314
機能コード……178
基本FCB……122
キャラクタ・デバイス……280
〈ク〉
国別情報……88,163
クラスタ……108
クラス名……50
グラフィック・コンソール……321
グラフLIO……315
繰り返しブロック……71
グループ……53
クロック・デバイス……287
〈ケ〉
ゲット・インタラプト・ベクタ法……293
〈コ〉
更新日時……112
コマンド・コード……282
コマンド・パケット……281,282
コマンド・プロセッサ……82,86
コントロール情報……101
コンソール……156
コンパイル……15
コンパイル・オプション……130
〈サ〉
再ロード……84
サブ・ディレクトリ……112
サブ・ファンクション……148
算術演算子……79
〈シ〉
シェル……82
シークレット……112
シーケンシャル・ファイル……122
システム・ファイル……112
実引数……69
シフト……65
シフト演算子……79
条件アセンブル……73
条件ディレクティブ……73
常駐……193
常駐終了……159
常駐部……84
除算エラー……210
〈ス〉
数値演算子……79
スタック・セグメント……49
スタック・フレーム……57
ステータス……283
ステートメント……32,34
ストラクチャ……64
ストラクチャ変数……64
ストラテジ……161

ストラテジ・ルーチン ……281
スモール・モデル ……105

〈セ〉

制御コード ……170
正常終了 ……95
セクタ ……109
セクタ・カウント ……287
セグメント ……30
セグメント演算子 ……76
セグメント・オーバライド・プレフィックス ……52
セグメント・ベース ……30
セグメント・レジスタ ……30
セグメント・ワード・サイズ ……49
セパレータ ……32

〈ソ〉

相対レコード ……123
ソース・ファイル ……32
ソート ……262

〈タ〉

代替マッピング・レジスタ ……310
タイプ・アヘッド・バッファ ……158
タイム・スタンプ ……233

〈チ〉

致命的エラー処理 ……131

〈テ〉

ディスク転送アドレス ……165
ディスク・バッファ ……86,120
ディスク・フォーマット ……108
ディレクティブ ……35
ディレクトリ ……108,306
ディレクトリ・エントリ ……109,111,112
データ定義 ……64
データの構造化 ……64
データ・ブロック ……64
デバイス属性 ……282
デバイス・タイプ ……178
デバイス・データ ……175
デバイス・ドライバ ……280
デバイス・ヘッダ ……280
デバイス名 ……282
デバッグ ……17
デリミタ ……32
転送アドレス ……286
テンポラリ・ファイル ……174

〈ト〉

トークン ……32
特殊デバイス ……146
特殊ファンクション ……178
ドライブ番号 ……122

〈ナ〉

内部割り込み ……126
名前 ……34

〈ネ〉

ネスト ……73

〈ハ〉

ハードウェア構成情報 ……310
バイト/セクタ・カウント ……287
バイナリ・ファイル ……16
パス名 ……122
バッファの数 ……86,121
パラグラフ ……88
パラメータ ……59
パラメータ・ブロック ……159,160,178
ハンドラ ……127
ハンドル属性 ……305
ハンドル名 ……305

〈ヒ〉

非常駐部 ……84
ビット・マスク ……65
標準入出力 ……85,124

〈フ〉

ファイル・サイズ ……113,123
ファイルの数 ……85
ファイルの属性 ……112
ファイルのベース名 ……112,122
ファイルのロック ……174
ファイル・ハンドル ……124,121
ファイル・ポインタ ……120,124,170
ファンクション コード ……288
フィールド ……34,64
ブート ……82
ブート・セクタ ……109
物理ページ ……292
ブレーク・アドレス ……284
プログラム・セグメント ……89,95
プロシージャ ……58
プロセス ……88
プロセスの終了 ……94
ブロック・デバイス ……280
ブロック・デバイス番号 ……285

〈ヘ〉

ページ ……101
ページ・フレーム ……292
ベース・アドレス ……64
ベース名 ……40
ヘッダ ……101
ベリファイ・フラグ ……162

〈ホ〉

補助出力 ……158
補助入力 ……158
ボリューム・ネーム ……112
ボリュームID ……286

〈マ〉

マクロ機能 ……67
マクロ定義 ……67
マクロ・ボディ ……67
マッピング ……292

〈メ〉
メディア交換……177
メディア・ディスクリプタ……285
メモリ管理情報……88
メモリ・ブロック……160
メモリ・マップ……82
メモリ・モデル……56,101
〈モ〉
モジュール……40
〈ユ〉
優先順位……79
ユニット・コード……288
ユニット数……282,284
〈ラ〉
ライブラリ……16
ラベル……58,61
〈リ〉
リアルタイム・クロック……161
リクエスト・ヘッダ……282
リターン・コード……95,159,160
リピート・ディレクティブ……71
リモート・ドライブ……180
リロケータブル……101
リロケート情報……101
リロケーション情報……101
リンク……16
リンク情報……282
〈ル〉
ルート・ディレクトリ……109
〈レ〉
レコード……64
レコード・サイズ……122,123
レコード長……283
レコード番号……108
レコード変数……64
レジスタ間接……78
〈ロ〉
ロウ・ページ……310
ローカス……164
ローカル・ドライブ……180
ロード・アドレス……31
ロード・モジュール……101
ロック……174
論理……109
論理演算子……79
論理セグメント……31,103
論理装置コード……283
論理ページ……292
〈ワ〉
ワイルド・カード……123
割り込みエントリ・ルーチン……281
〈A〉
Abort……132
align……41
ASCIZ 文字列……97
ASSUME……52
ASSUME ディレクティブ……52
AT……49
autoexec.bat……84
〈B〉
BIOS……83
BPB……285
BPB テーブル……108
BPB 配列……284,285
BREAK コマンド……85
BREAK チェック・フラグ……163
BUFFERS コマンド……86,121
BUILD BPB……286
BUSY ビット……284,287
BYTE……41
〈C〉
class……50
CodeView……23
config.sys……84
COM モデル……105
combine……42
command.com……82
COMMON……45
COMSPEC……87
COPY キー……193
COUNTRY コマンド……87
CP/M コンパチブル……148
CREF……20
ctrl-C 処理ルーチン……127
ctrl-C 入力……156
〈D〉
Define……64
Define ディレクティブ……64
DEVICE コマンド……85,285
DEVICE OPEN/CLOSE……288
DONE ビット……283
DTA……95,97,165
DUP 演算子……78
DWORD……41
〈E〉
EDLIN……18
ELSE ディレクティブ……73
EMM ハンドル……294
EMS……291
ENDM ディレクティブ……67
ENDP ディレクティブ……58
ENDS ディレクティブ……41
ERR ビット……284
EXE モデル……101
EXE2BIN……20

EXEC システム・コール ……89
EXTRN ……61
EXTRN ディレクティブ ……61

〈F〉

Fail ……132
FAT ……108,113
FAT エントリ ……110,112,114
FAT-ID ……114,285
FCB ……86,95,121,166
FCBS コマンド ……86
FILES コマンド ……85
FLUSH ……287

〈G〉

GROUP ……53
GROUP ディレクティブ ……53
Generic IOCTL ……288

〈I〉

IF ディレクティブ ……73
Ignore ……132
INIT ……284
INT 00H ……211
INT 05H ……193
I/O リクエスト ……283
IOCTL データ ……176
io.sys ……82
IPL ……83,109
IRP ……71
IRP ディレクティブ ……71
IRPC ……71
IRPC ディレクティブ ……71

〈L〉

LABEL ……61
LABEL ディレクティブ ……61
LASTDRIVE コマンド ……86
LENGTH ……78
LENGTH 演算子 ……78
LIB ……20
LINK ……19

〈M〉

MACRO ……67
MAKE ……26
MAPSYM ……21
MASM ……18
MEDIA CHECK ……285
MEMORY ……49
msdos.sys ……82

〈N〉

NAME ディレクティブ ……40
NEAR ……31
NEAR ラベル ……61
NON-DESTRUCTIVE ……287
NO-WAIT ……287

〈O〉

OFFSET ……78
OFFSET 演算子 ……78
OS/E ……293

〈P〉

PAGE ……42
PARA ……42
PRIVATE ……42
PROC ……58
PROC ディレクティブ ……58
PRT ……78
PRT 演算子 ……78
PSP ……88,95
PUBLIC ……43,61
PUBLIC ディレクティブ ……61
PURGE ディレクティブ ……69

〈R〉

RECORD ……65
RECORD ディレクティブ ……65
REMOVABLE MEDIA ……288
REPT ……71
REPT ディレクティブ ……71
Retry ……132

〈S〉

SEG ……78
SEG 演算子 ……78
SEGMENT ……41
SHELL コマンド ……86
SIZE ……78
SIZE 演算子 ……78
STACK ……48
STATUS ……287
STRUC ディレクティブ ……64
SYMDEB ……22
sysinit ……83

〈T〉

TPA ……95
TSR ……193
TYPE ……78
TYPE 演算子 ……78

〈U〉

use ……49

〈W〉

WORD ……41

〈記号〉

－As オプション ……130
－Gs オプション ……135
－J オプション ……130
/ML オプション ……130
.MODEL ……56
.MODEL ディレクティブ ……55
.TYPE ……78
.TYPE 演算子 ……78

図表索引

エラー・コード(表8-2)……284
拡張メモリ・マネージャのステータス・コード(表9-2)300
拡張 FCB のフォーマット(図4-12)……123
基本 FCB のフォーマット(図4-9)……122
グラフ LIO の入出力パラメータ(表A-1)……315
コマンド・パケット(図8-5)……283
ステータス・フィールドの各ビットの意味(図8-6)……284
属性のビット(表4-2)……112
ディレクトリ・エントリのフォーマット(図4-4)……112
デバイス・コマンド・コード(表8-1)……283
デバイス属性フィールドの各ビットの意味(図8-4)……282
デリミタとセパレータの一覧(表2-1)……33
日付けおよび時刻の記録フォーマット(図4-5)……113
ファンクション・リクエスト一覧(表6-1)……148
ヘッダのコントロール情報(表3-5)……102
メモリ管理情報(図3-4)……89
AXレジスタの内容(図5-1)……131
CodeView の起動オプション(表1-9)……24
CodeView のダイアログ・コマンド(表1-10)……24
DTA 内フォーマット(図6-6)……172
DX レジスタに返されるデバイス情報(図6-7)……176
EDLIN のサブコマンド(表1-4)……18
EMM ファンクション一覧(表9-1)……296
LINK の起動オプション(表1-6)……21
MAKE の起動オプション(表1-11)……27
MASM の演算子(表2-6)……77
MASM の起動オプション(表1-5)……19
MASM のディレクティブ一覧(表2-2)……36
MS-DOS の主なディスク・フォーマット(表4-1)……108
MS-DOS ver.3.3 の主要コマンド(PC9801 用システム・ディスク)(表1-2)……12
PSP の構成(図3-8)……95
SYMDEB のサブ・コマンド(表1-8)……22
ver.2.11 以降のファンクション・リクエストにおけるエラー・コード(表6-2)……155

● ディスク・サービス申し込み用紙

I/F MS-DOS

ご住所：〒

会社名：

所属部課名：

お名前：　　☎（　　）　　☎（　　）

▼送付先の住所と宛名を記入してください(送付ラベル用)

〒

右記の"ディスク・サービスについて"を確認のうえ，2箇所に住所・氏名を記入して，代金3,000円と共に現金書留でお送りください．会社の方は会社名と所属部課名も記入してください．

〈宛先〉 〒170 東京都豊島区巣鴨1-14-2
CQ出版社 C&E出版部
MS-DOSプログラマーズ・バイブル係

ディスク・サービスについて

本書で紹介したプログラムのソース・ファイル，および実行形式ファイルをフロッピ・ディスクで頒布いたします．ご希望の方は，申し込み用紙に必要事項を記入して，代金とともに現金書留でお送りください．整理の都合上，締切は毎月月末とし，締切後2週間以内に発送いたします．（申し込み期限：1990年3月末日）

● **ディスクの内容**

本書の第5章，第7章，および Appendix B/C で解説したサンプル・プログラムのソース・ファイル（*.C および *.ASM）と実行形式ファイル（*.COM あるいは *.EXE）．

● **メディアの種類**

MS-DOS版5.25インチ2DDタイプ・フロッピ・ディスク．

● **代金**

3,000円（消費税・送料込み）

● **宛先**

〒170 東京都豊島区巣鴨1-14-2
CQ出版㈱ C&E出版部
MS-DOSプログラマーズ・バイブル係

インターフェース増刊

MS-DOS プログラマーズ・バイブル

定価は表四に表示してあります

1989年10月15日 発行

著者 阿部英志
発行人 神戸一夫
発行所 CQ出版株式会社
〒170 東京都豊島区巣鴨1－14－2
☎(03)947－6311(代表) 振替 東京0-10665

写植／みやこワードシステム 印刷・製本／三共グラビヤ印刷(株)

漢字ターミナル Ver.3.10

パソコンを端末に利用できます!

ワークステーションを使っているのだが、端末がもう1台ほしいと思ったことはありませんか?もし貴方の利用しているOSがUNIXかOS-9/68Kならば、パソコンを端末に利用しましょう!このユーティリティーは、VT100とほぼ同等の機能に加えファンクションキーの設定複数ファイルの転送など、多数機能をもっています。

PC9801、PC-286シリーズ用(NSF1010)	¥12,800

制御屋の作ったアセンブラをプロの貴方に!

"新しいCPUを使ってみたいのだがアセンブラが……"と思ったことは、ありませんか? 4bits, 8bitsのCPUならこのアセンブラ1つでOKです。新たにクロスアセンブラを購入する必要はありません。標準で8085、Z80、6301、6805、6809、68HC11、Melps740、uPD7500の命令定義ファイルを添付。これにより、標準でアセンブルできるCPUが30を超えます。ユーザーの命令定義も豊富な定義文により可能です。未解決シンボル、アドレス表示を絶対値に変換するリストコンバータを新たに付属しました。

OS-9/680X0	¥238,000	MS-DOS	¥218,000
OS-2	¥228,000	UNIX	¥600,000

バージョン3.10新発売!!

〈バージョンアップ受付中〉

OS-9/68000ユーティリティー

[1]PC9801-29(GPIBボード)用 ドライバーソフト

ソースコード(NSF-1011)	¥150,000
オブジェクトコード(NSO-1002)	¥50,000

フォークス製OS-9(PC9801用)でGPIBボードを利用するためのソフトです。

[2]プログラム開発ユーティリティセット

(NSO-1101)	¥19,800

xref 簡易クロスリファレンス
more UNIX moreのサブセット
rm 最後にデリートしたファイルを復活できるデリートプログラム
cat ファイル連結、追加プログラム

各ユーティリティープログラムは、
下記のところでもお求めになれます。

株式会社フォークス
(クロスアセンブラーMS-DOS版を除く)
東京都中央区八丁堀4-14-1 Tel. 03-553-4911

有限会社メディア・コム
奈良県天理市檜垣町99-2 Tel. 07436-6-3096

※OS-9はマイクロウェア社の登録商標です。
※MS-DOSはマイクロソフト社の登録商標です。
※UNIXはベル研究所で開発されたOSの名称です。

Nss Ltd.,

MICROCOMPUTERS SYSTEM & SOFTWARE

有限会社 エヌ・エス・エス 〒639-11 奈良県大和郡山市矢田山町13-9 TEL 07435-2-6809(代表) FAX 07435-2-8086

インターフェース増刊　MS-DOS プログラマーズ・バイブル　CQ出版社　定価2,500円(本体2,427円)　雑誌 01620-10